MASTER EN SCIENCES INFORMATIQUES

La détection en temps réel de collision entre objets dans les environnements virtuels

Bergeret, Lionel

Award date:
2000

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année Académique 1999-2000

La détection en temps réel de collision entre objets dans les environnements virtuels

Lionel **BERGERET**

Promoteur : Professeur Jean-Paul Leclercq

Mémoire en vue de l'obtention du grade de Maître en Informatique

Résumé

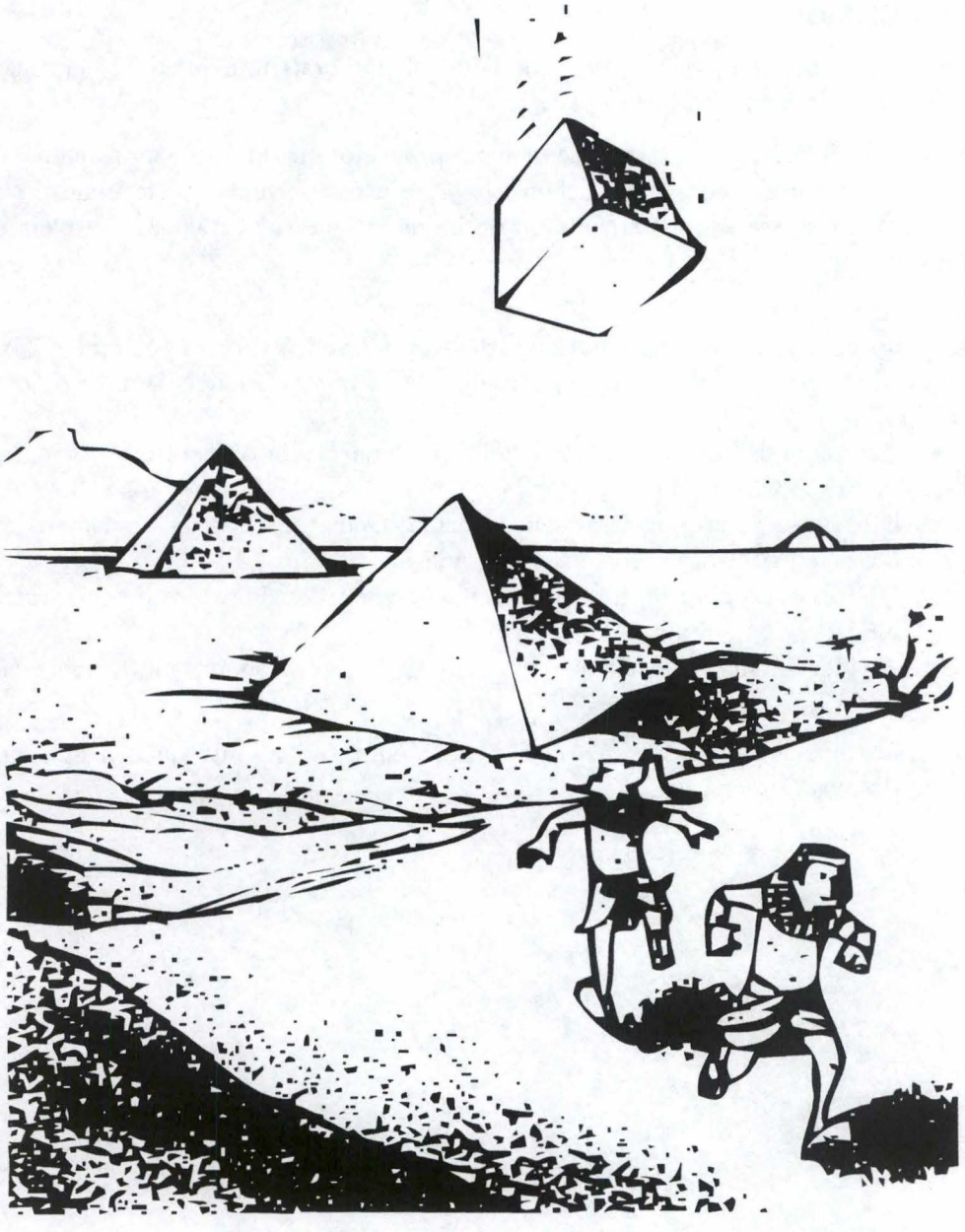
Dans ce mémoire, nous allons investiguer et analyser les divers mécanismes de détection de collision dans les environnements virtuels qui requièrent le "temps réel".

Les objets solides dans le monde réel ne se traversent pas entre eux lorsqu'ils entrent en collision. Attribuer cette propriété de "solidité" aux objets est d'une importance capitale dans beaucoup d'applications graphiques interactives. Par exemple, la solidité rend les mondes virtuels plus réalistes, et elle est également essentielle pour les simulateurs de véhicules. Ces applications utilisent un algorithme de détection de collision pour donner de la solidité aux objets qu'elles manipulent. Pour fonctionner dans ces applications, un algorithme de détection de collision doit fournir des réponses en temps-réel, même quand beaucoup d'objets sont en jeu, et il doit supporter des objets dont les mouvements sont spécifiés sur le tas par un utilisateur.

Abstract

In this thesis, we investigate and analyse several mechanisms of collision detection in virtual environments, which require real-time rates.

Solid objects in the real world do not pass through each other when they collide. Enforcing this property of "solidness" is of paramount importance in many interactive graphics applications; for example, solidness makes virtual reality more believable, and solidness is essential for the vehicle simulators. These applications use a collision-detection algorithm to enforce the solidness of objects. To work in these applications, a collision-detection algorithm must run at real-time rates, even when many objects can collide, and it must tolerate objects whose motion is specified "on the fly" by a user.



"Look out! Here comes another one!"

Remerciements

Je tiens à exprimer ma gratitude envers Tri Vu Khac pour m'avoir incité et encouragé à réaliser ce sujet de mémoire fort captivant.

Je tiens à remercier tout particulièrement mon promoteur, le professeur Jean-Paul Leclercq, ainsi que le professeur Jean Fichefet pour m'avoir permis d'entreprendre ce sujet de mémoire très intéressant qui touche à un domaine qui me tient à coeur, à savoir l'informatique graphique.

Je remercie également :

- Marc Luppi, physicien et informaticien, développeur chez Virtual Technology, Namur.
- Dinesh Manocha, professeur au Department of Computer Science, University of North Carolina.
- Stefan Gottschal, informaticien, chercheur au Department of Computer Science, University of North Carolina.
- James Klosowski et Joseph Mitchell, informaticiens, professeurs au Department of Computer Science, State University of New York at Stony Brook.
- Jean-Christophe Lombardo, informaticien, chercheur sur le projet de chirurgie hépatique, INRIA, site de Sophia-Antipolis
- Lisette Calderan, informaticienne, INRIA, SICS Département multimédia (Photothèque)

Enfin et surtout, je voudrais remercier mes parents, qui m'ont soutenu et encouragé tout au long de mes études.

Table des matières

Table des figures

Introduction

Un objectif important en informatique graphique est de produire des animations qui reproduisent le monde physique de manière plausible. Beaucoup de facteurs contribuent à rendre une animation vraisemblable. La modélisation géométrique des objets tridimensionnels et le rendu de ces objets 3D en images bidimensionnelles doit être réaliste, un grand nombre de travaux ont déjà été faits dans ce domaine. Un autre facteur tout aussi important est le comportement des objets lorsqu'ils se déplacent à travers les images fixes[1] d'une animation. Donner un comportement réaliste aux objets d'une animation pose encore des défis à la recherche.

Un défi parmi les plus importants implique notamment les collisions entre objets. Dans beaucoup de situations, une animation sera d'autant plus réaliste si les objets ne se traversent pas les uns les autres. Ce ne sera le cas que si les objets présentent de la *solidité.*

Certains programmes d'animation peuvent demander à l'utilisateur d'appliquer manuellement cette solidité à leurs objets. Ceci implique que l'utilisateur doit explicitement décrire le mouvement de chaque objet pour chaque image de l'animation. Dans ce cas, l'utilisateur est responsable de savoir quand les objets vont entrer en collision et ce qu'il se produit après chaque collision. Dans beaucoup de contextes, l'application manuelle de la solidité aux objets devient vite infaisable. En effet, les systèmes de simulation réaliste, créant des représentations de parties mécaniques, d'outils, de machines pour tester leurs interconnectivités, leurs fonctionnalités et leur fiabilité, requièrent de modéliser l'interaction entre les objets de manière très précise. Ces applications ont besoin d'un moyen pour appliquer la solidité de manière automatique.

Détecter les collisions et déterminer les points de contacts est un des points cruciaux pour une description fidèle des interactions entre les objets dans une simulation. Ce type d'application peut compter sur un système de prise en charge des collisions pour donner de la solidité aux objets qu'il manipule. Ce système peut être séparé en trois parties : la détection des collisions, la détermination des zones de contact et la réponse aux collisions [2]. Les deux premiers points sont souvent mis ensemble pour former l'algorithme de détection. Il a pour but de détecter les objets qui commencent à se pénétrer les uns les autres et ensuite de fournir les zones [3] de contacts. Si l'algorithme de détection trouve de tels objets

[1] "frames" en anglais

[2] Ceci peut être, bien évidement, mis au singulier si seulement deux objets s'interfèrent

[3] suivant les besoins, les zones signifient les points, les arêtes, les faces ou autre chose

alors le système de prise en charge des collisions appelle son troisième composant qui est l'algorithme de réponse. Celui-ci à pour objectif de corriger le comportant des objets entrant en collision d'après leur masse, leur vélocité, etc. afin qu'ils ne se pénètrent plus entre eux. Cette correction, par exemple, peut faire qu'un objet rebondisse sur un autre.

Aussi bien la détection de collision que la réponse aux collisions posent d'intéressants problèmes à la recherche. Le domaine étant très vaste sur les moyens utilisés (programmation linéaire, etc.), nous nous limiterons, si on peut considérer cela comme une limite, aux applications exigeants des réponses en temps-réels. Les résultats obtenus sont, bien évidemment, utilisés pour d'autres types d'applications qui ont besoin de beaucoup plus de précision (les crash-test pour voiture, etc.) sans demander le temps-réel.

Une partie de plus en plus importante des applications actuelles produit des animations dites *interactives*. Dans une application interactive, l'utilisateur contrôle directement le déroulement des événements et reçoit immédiatement une réponse de celle-ci. Par contraste avec les applications recevant une séquence de commandes fournies par l'utilisateur et ne produisant aucun résultat tant que toute la séquence n'ait été reçue. On peut constater que dans beaucoup de situations, les utilisateurs trouvent le répondant des applications interactives plus naturel et surtout plus confortable.

Les applications interactives partagent deux caractéristiques particulières :
- l'application ne peut connaître exactement ce qui va ce produire dans le futur. Cette incertitude vient du fait que les utilisateurs déterminent le déroulement des événements de l'application sur le tas.
- l'application doit répondre aux actions des utilisateurs en temps réel.

Pour une application produisant une animation, un élément important pour une performance temps réel est de maintenir un niveau élevé et presque constant du nombre d'images par seconde [4]. En général, le but à atteindre doit varier entre 20 et 30 images par seconde. Un autre aspect important pour une performance temps réel est la faible latence, aussi appelé retard. La latence est le temps entre le moment où un utilisateur déclenche une action et le moment où l'utilisateur observe la manifestation de l'action dans l'application. Ce temps n'est pas défini précisément, mais des recherches ont montré qu'une latence supérieure à 100 ms devient vite inconfortable pour les utilisateurs.

Une variété d'applications interactives a besoin de détecter les collisions. Les simulateurs de véhicule, comme les simulateurs d'avions ou les simulateurs de conduite, en sont un exemple. Ces types de simulateurs sont importants parce qu'ils entraînent les utilisateurs à manoeuvrer des véhicules, parfois compliqués, sans risquer de les blesser ou de causer des dégâts aux véhicules. Tant que les utilisateurs auront besoin de piloter des véhicules sans se heurter à d'autres objets, les détections de collisions seront importantes dans ces simulateurs.

Une autre exemple d'application est la modélisation moléculaire. Les effets d'une molécule sont déterminés, en partie, par la façon dont elle s'adapte géométriquement dans

[4] généralement noté "fps" en anglais (frames per second)

un emplacement récepteur. Les simulations par ordinateur qui permettent aux chimistes de tester interactivement comment les molécules s'assemblent les unes aux autres sont des outils précieux dans la conception de nouveaux médicaments. Détecter les collisions entre les molécules est une part importante de ce type de simulations.

Un type d'applications qui devient de plus en plus populaire est la réalité virtuelle. Dans ces applications, l'ordinateur essaye de donner aux utilisateurs le sentiment d'être plongé dans un nouvel environnement. Ils leur est possible, par exemple, de saisir ou pousser des objets. Les utilisateurs peuvent perdre ce sentiment de réalité si les objets les entourant dans ce nouveau monde se comportent d'une manière peu naturelle. Le fait que des objets se traversant entre eux n'est pas naturel dans la plupart des situations. Il est important ici aussi de disposer d'un mécanisme de détection de collisions. En général, une détection de collisions avec une réponse appropriée peut faire qu'une application de réalité virtuelle soit plus réaliste (ce qui est censément pourquoi on l'appelle la "réalité" virtuelle), parce que cette détection est la première étape vers un comportement des objets plus "réel".

La réalité virtuelle a longtemps été considéré comme un jouet plutôt qu'un outil. Beaucoup de compagnies ont commencé à étudier la réalité virtuelle comme outil pour évaluer les maquettes numériques. Une des fonctions principales nécessaire pour l'évaluation interactive est la détection en temps-réel des collisions.

Le prototypage virtuel (les maquettes numériques) est devenu de plus en plus important pour aider à réduire le temps de mise sur le marché et, par conséquent les coûts d'un nouveau modèle ou produit. Pour exemple, chaque jour de retard, dans la production d'un nouveau modèle de voiture, coûte en moyenne environ 80 millions de francs belges [Zac97]. Beaucoup de compagnies, particulièrement dans le domaine de l'automobile et dans l'industrie aérospatiale ont commencé à évaluer la réalité virtuelle (VR) comme outil principal pour la DAO[5] et l'IAO[6] afin d'étudier la maquette numérique d'une nouvelle conception. L'idée est de permettre aux concepteurs, planificateurs de fabrications, stylistes, et analystes d'évaluer plusieurs aspects du nouveau produit interactivement et en totale immersion. Par exemple, la vérification d'assemblage et de désassemblage. Lors de cette vérification des questions doivent être posées, est-ce que l'assemblage est difficile ou facile ? Combien de temps cela prend ? Y a-t-il assez de place pour les outils ? etc.

Dans ce mémoire, nous allons investiguer et analyser les divers mécanismes de détection de collision dans les environnements virtuels qui requièrent le "temps réel".

Nous commencerons dans le chapitre 1 par définir les connaissances de base de l'informatique graphique nécessaire pour mieux aborder la suite. Ensuite dans le chapitre 2, nous allons faire une classification des différentes questions qu'il faut se poser avant d'entreprendre l'analyse proprement dite du problème de la détection de collision. Nous ferons dans le chapitre 3 un petit tour des principales structures de données permettant de stocker un objet graphique. Les chapitres 4 et 5 seront entièrement consacrés à la résolution du

[5]dessin assisté par ordinateur (abréviation CAD en anglais).

[6]ingénierie assistée par ordinateur (abréviation CAE en anglais). Pour une terminologie complète des différents termes utilisés en informatique graphique, rapportez-vous à l'annexe A

Chapitre 1

L'informatique graphique, un aperçu

1.1 Introduction

Les ordinateurs actuels ayant dans beaucoup de cas l'écran comme dispositif principal de sortie, le rôle des images produites et affichées est de plus en plus important ces dernières années. La place des outils mathématiques employés dans ce domaine y est devenue prépondérante, sans que les utilisateurs en perçoivent toujours tous les aspects.

L'informatique graphique[1] recouvre plusieurs notions qu'on peut classer sommairement de la façon suivante, bien que bon nombre de logiciels fassent appel à une combinaison de différentes techniques :

- la *modélisation géométrique* concerne les propriétés des courbes, surfaces et polyèdres.
- la *géométrie algorithmique* est l'ensemble des méthodes qui permettent d'effectuer, par algorithmes, des opérations sur des objets géométriques : intersections, triangulations, enveloppes convexes,...
- la *synthèse d'images* permet de produire des images à partir de modèles. Les calculs sont faits en prenant en compte un modèle d'éclairement, local ou global, qui peut tenir compte de l'optique géométrique : réflexion, transmission, absorption, coefficients de Fresnel, mais aussi ajouter des textures 2D ou 3D, déterministes ou aléatoires, avec des modèles empiriques d'ombrage.
- le *traitement d'images* autorise des modifications automatiques sur des images en général acquises par mesures : segmentation, filtrage, reconnaissance de formes.

Nous n'aborderons dans ce mémoire que les deux premiers points. Sur le plan pratique se posent aussi nombre de problèmes d'implémentation, de nature plus spécifiquement informatique : structures de données, gestion de bases de données, complexité.

[1] le terme *infographie* est souvent utilisé à la place

problème de la détection entre paire d'objets, d'abord de manière algorithmique puis en exploitant les ressources matérielles de l'ordinateur.

Nous terminerons par le chapitre 6 dans lequel nous verrons quelques unes des extensions que nous pouvons apporter aux algorithmes de détection de collision entre paire d'objets, notamment, la prise en charge des objets non-convexes, la gestion d'un environnement multi-objets nécessitant des mécanismes de traitement plus élaborés, etc.

1.2 Définitions

Nous allons donner la définition de quelques concepts importants pour mieux appréhender les problèmes que nous aborderons dans la suite. Nous allons voir la définition

- de la convexité,
- d'un polygone,
- d'un polyèdre,
- d'un polytope,
- de ce qu'est une enveloppe convexe,
- de ce que sont les diagrammes de Voronoï
- de volume limite.

1.2.1 Convexité

Un ensemble S de points est *convexe* si pour tous points $a, b \in S$, nous avons que le segment $ab \subseteq S$, c'est-à-dire

$$\forall a, b \in S \quad \{x : x = \lambda a + (1 - \lambda)b \text{ où } \lambda \in [0, 1]\} \subseteq S$$

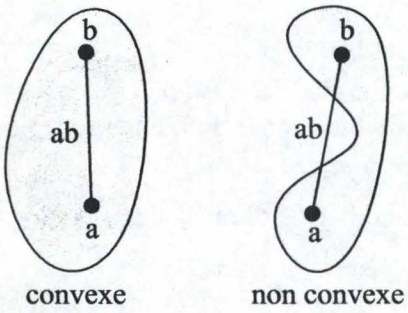


FIG. 1.1 – Ensemble de points formant une région de l'espace convexe et une non-convexe

Une combinaison linéaire de points x_i $(i = 1, .., k)$ appartenant à un ensemble de points S est convexe si et seulement si

$$\begin{cases} \sum_{i=1}^{k} \lambda_i x_i \in S \\ \forall \lambda_i \geq 0, \sum_{i=1}^{k} \lambda_i = 1 \end{cases}$$

Cela signifie qu'un segment de droite consiste en toutes les combinaisons convexes de ces points extrémaux, et un triangle consiste en toutes les combinaisons convexes de ces trois sommets. En trois dimensions, un tétraèdre est la combinaison convexe de ces quatre sommets.

1.2.2 Polygone

Soient $v_0, v_1, v_2, ..., v_{n-1}$ n points dans le plan. Tout les indices sont modulo n, impliquant un ordre cyclique des points, avec v_0 suivant v_{n-1}, en effet $(n-1)+1 \equiv n \equiv 0(mod\ n)$. Soient $e_0 = v_0v_1, ..., e_i = v_iv_{i+1}, ..., e_{n-1} = v_{n-1}v_0$ n segments connectant les points. Ces segments forment un polygone ssi

1. l'intersection de chaque paire de segments adjacents, pris dans l'ordre cyclique, est égal au seul point qui est partagé entre eux : $e_i \cap e_{i+1} = v_{i+1}, \forall i \in \{0, ..., n-1\}$.
2. les segments non adjacents ne se coupent pas : $e_i \cap e_j = \emptyset, \forall j \neq i+1$.

Les points v_i sont appelés les *sommets* et les segments e_i sont appelés les *arêtes*. Nous remarquons aussi qu'un polygone qui a n sommets a n arêtes.

Un polygone *régulier* est un polygone qui a ses côtés égaux et ses angles égaux. Le triangle équilatéral, carré, pentagone régulier, hexagone régulier, etc. en font partie.

1.2.3 Polyèdre

Un polyèdre est la généralisation naturelle d'un polygone bidimensionnel vers la troisième dimension. Sa surface est composé de trois types d'objets géométriques :

1. sommets 0-dimensionnels (points)
2. arêtes 1-dimensionnelles (segments)
3. faces 2-dimensionnelles (polygones)

Une simplification utile serait que les faces soient des polygones *convexes*. Ceci est sans perte de généralité puisque toute face non-convexe peut être partitionnée en faces convexes, ces faces étant adjacentes et coplanaires.

En résumé, un polyèdre est constitué d'une collection finie de faces polygonales convexes coplanaires tel que

1. les faces se coupent "proprement", c'est-à-dire que pour chaque paire de face, il est requit que
   - elles soient disjointes, ou
   - elles aient un seul somment en commun, ou
   - elles aient deux sommets, et l'arrête les reliant, en commun.
2. le voisinage de chaque point frontière du polyèdre est topologiquement un disque ouvert (figure 1.2), c'est-à-dire, "homéomorphe"[2] à un disque. Un homéomorphisme entre deux régions autorise les étirements et pliures mais pas les déchirures.
3. la surface est connectée et fermée. De cette façon, il est possible de n'importe quel point du polyèdre de se déplacer vers un autre en "marchant" sur la surface.

[2]deux régions sont homéomorphe s'il y a une projection topographique (mapping) entre eux qui soit un-à-un et continue ([O'R98] p. 102).

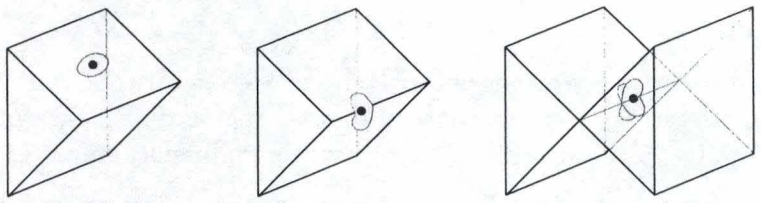


FIG. 1.2 – Chaque point frontière a un voisinage de points l'entourant qui forme un disque topologique (a). En (b), ce point n'a pas un voisinage qui est un disque topologique.

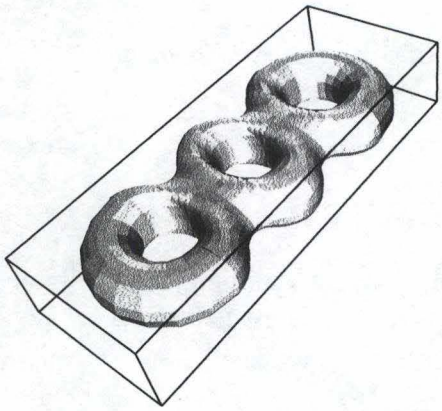


FIG. 1.3 – Polyèdre de genre 3

De plus, nous pouvons ajouter à cette définition la notion de trou, dans le sens d'un tunnel partant d'un côté de la surface du polyèdre et allant vers l'autre côté sans déconnecter l'extérieur, c'est-à-dire qu'un segment de droite, dont les extrémités sont des points extérieurs au polyèdre, traversant ce tunnel ne sera pas en intersection avec le polyèdre. Le nombre de trou est appelé le *genre* de la surface. Nous permettons les polyèdres à avoir un nombre arbitraire de trous (figure 1.3). Ainsi, un tore est, malgré un trou, considéré comme un polyèdre de genre 1.

La frontière d'un polyèdre est, comme nous venons de le définir, fermée et donc enferme une région de l'espace. Chaque arête est partagée par exactement deux faces, c'est faces sont appelées *adjacentes.*

Léonard Euler remarqua une étonnante régularité dans les nombres de sommets, arêtes et faces d'un polyèdre de genre zéro :

$$S - A + F = 2$$

où S est le nombre de sommets, A le nombre d'arêtes et F le nombre de faces.

1.2.4 Polytope

Un polytope est un polyèdre de genre zéro convexe, c'est-à-dire que tout segment reliant deux de ses sommets est entièrement contenu dans la région de l'espace qu'il délimite. Il est parfois noté 3-polytope pour mettre l'accent sur sa tridimentionnalité.

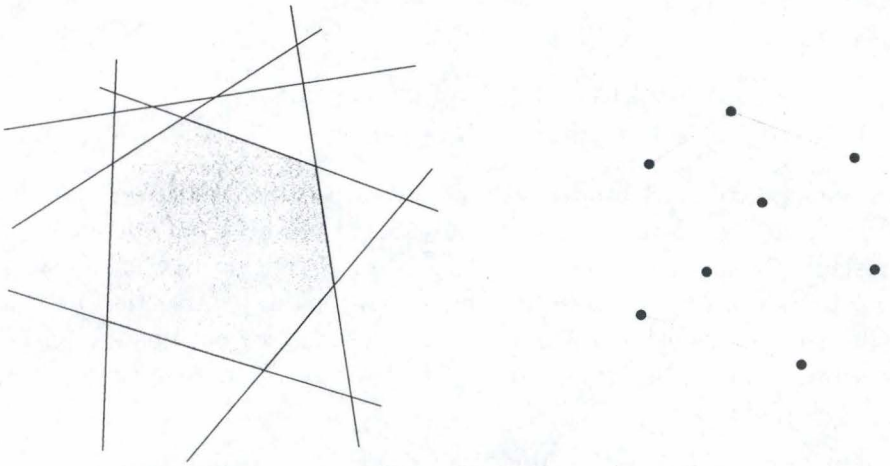
Un polytope constitué de polygones réguliers est appelé un polytope régulier. Une implication surprenante de cette régularité est qu'il n'y a seulement cinq types distincts de polytopes réguliers (tétraèdre, cube = hexaèdre, octaèdre, dodécaèdre et icosaèdre). Ils sont connus sous le nom de solides platoniques.

La formule d'Euler est vérifiée pour eux également :

Nom	S	A	F
tétraèdre	4	6	4
cube	8	12	6
octaèdre	6	12	8
dodécaèdre	20	30	12
icosaèdre	12	30	20

1.2.5 Enveloppe convexe

L'enveloppe convexe d'un ensemble de points A est l'intersection bornée d'un nombre fini de demi-espaces qui contient A. Elle est généralement noté $conv(A)$.



(a) (b)

FIG. 1.4 – (a) l'intersection bornée d'un nombre fini de demi-espaces et (b) l'enveloppe convexe résultante (2D)

L'enveloppe convexe $conv(A)$ peut être aussi définie comme étant
- l'ensemble de toutes les combinaisons linéaires convexes des points de A,
- le plus petit ensemble convexe contenant A,

– l'intersection de tous les convexes contenant A.
Ces définitions sont toutes équivalentes à la première vue ci-dessus.

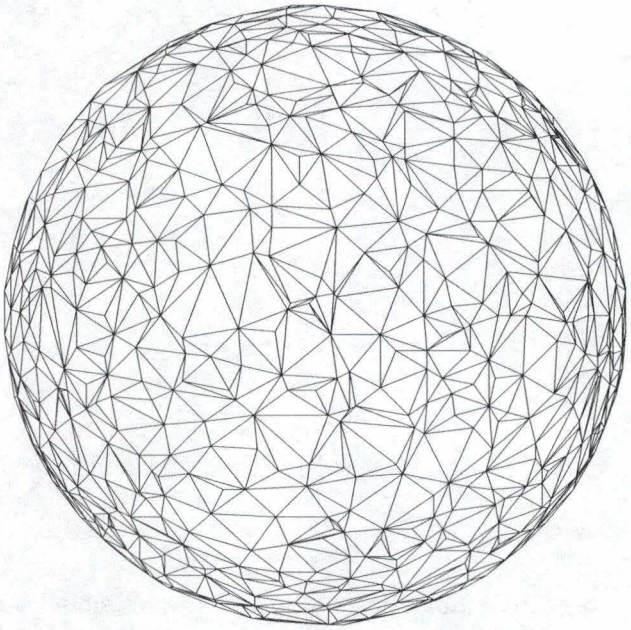


FIG. 1.5 – L'enveloppe convexe d'un ensemble de 1000 points générés aléatoirement à l'intérieur d'une sphère (Vue frontale)

1.2.6 Diagrammes de Voronoï

Soit S un ensemble de m sites, notés s_i, de l'espace euclidien en dimension n. Pour chaque site s_i de S, la cellule de Voronoï V_i de s_i est l'ensemble des points de l'espace qui sont plus proches de s_i que de tous les autres sites s_j $(j \neq i)$ de S, c'est-à-dire, $V_i = \{x : \forall j \neq i \, \|x - s_i\| \leq \|x - s_j\|\}$. Le diagramme de Voronoï de $V(S)$ est la décomposition de l'espace formée par les cellules de Voronoï des sites (figure 1.6 (a)).

Le cas classique est celui où les s_i $(i = 1, .., m)$ sont des points de $\mathbb{R}^n$ est où d est la distance euclidienne, les lignes à égale distance des sites sont alors les médiatrices des segments qui les joignent et les V_i $(i = 1, .., m)$ sont des polygones convexes si on est dans le plan, des polyèdres convexes en dimension supérieure.

On associe à un diagramme de Voronoï classique $V(S)$ une triangulation $D(S)$ dite de Delaunay par dualité de graphe : deux points de S sont reliés par une arête dans la triangulation de Delaunay si et seulement si leurs cellules sont adjacentes dans le diagramme de Voronoï de S (figure 1.6 (b)).

Les diagrammes de Voronoï sont des structures très utiles, rencontrées fréquemment car elles permettent de représenter des relations de distance entre objets et des phénomènes

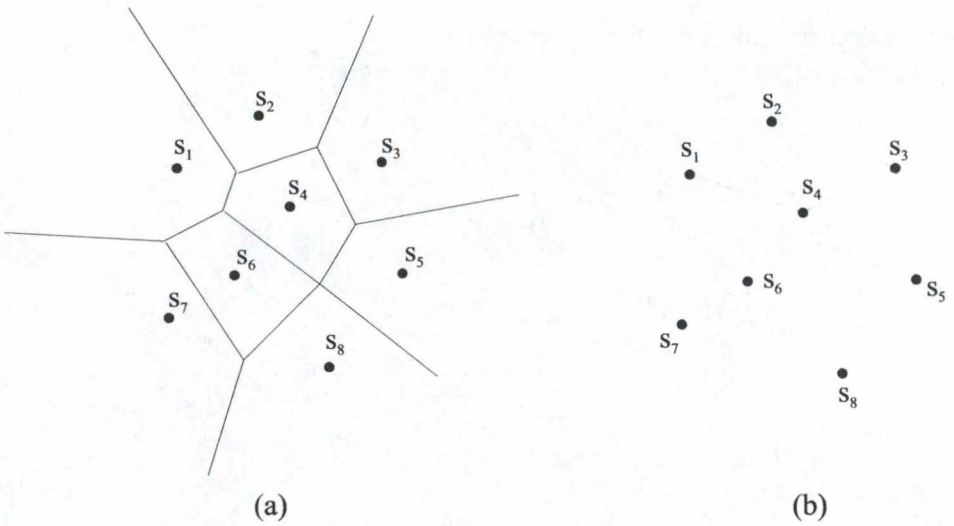


FIG. 1.6 – (a) le diagramme de Voronoï et (b) la triangulation de Delaunay d'un ensemble de points (2D)

de croissance : il n'est pas étonnant de les voir utilisés pour modéliser des cristaux ou les grandes structures de l'univers, et de les trouver souvent dans la nature, par exemple sur la carapace d'une tortue ou sur le cou d'une girafe réticulée. Les diagrammes de Voronoï sont aussi des structures de données permettant de résoudre de nombreux problèmes comme la recherche de plus proches voisins et la planification de mouvements (figure 1.8). L'étude des diagrammes de Voronoï, de leurs propriétés mathématiques, de leur calcul et de leurs nombreuses variantes a été et reste un sujet d'importance majeure de la géométrie algorithmique.

1.2.7 Volume limite

Le volume limite (BV) d'un objet (appelé parfois volume enveloppe, volume englobant), représenté par un ensemble de points, est un volume convexe fermé, dont la forme et l'orientation sont fixées, qui contient entièrement l'objet auquel il est associé, c'est-à-dire, tous les points de l'objet sont contenus à l'intérieur. Les formes des volumes limites les plus répandues sont les sphères et les boîtes (parallélépipèdes rectangles). Nous pouvons rencontrer différentes variantes de boîtes limites suivant leur orientation, notamment les boîtes limites à axes alignés (AABB) et les boîtes limites orientées (OBB). Nous aurons l'occasion d'en reparler dans le chapitre 4.

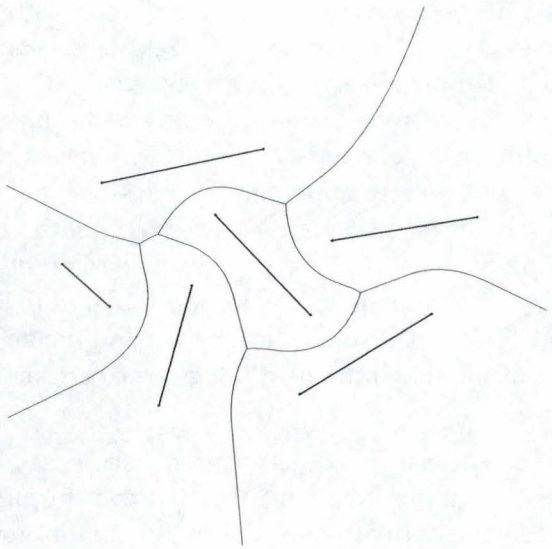


FIG. 1.7 – Diagrammes de Voronoï de segments de droite

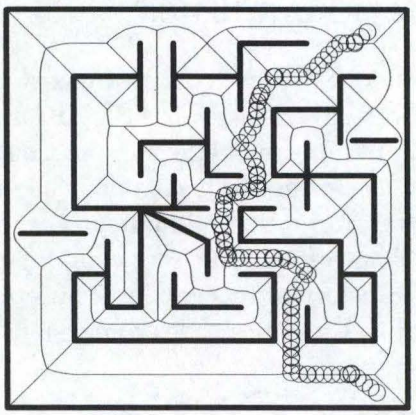


FIG. 1.8 – Application à la planification de trajectoires : déplacement d'un disque

1.3 Les transformations

Les transformations géométriques sont des outils très importants pour la génération de scènes tridimensionnelles. Elles sont utilisées pour déplacer des objets à l'intérieur d'un environnement et aussi pour construire une vue en deux dimensions de l'environnement en vue d'un affichage sur écran.

En informatique graphique, la méthode la plus fréquente pour représenter des objets est

l'utilisation d'un ensemble de polygones. Nous pouvons représenter un objet comme une liste de points dans un espace à trois dimensions. Nous faisons cela en représentant la surface d'un objet par un ensemble de polygones connectés entre eux, où chaque polygone est une liste de points. Cette forme de représentation est soit exacte ou soit approximative, ceci dépend de la nature de l'objet. Un cube peut être représenté exactement grâce à six carrés. Par contre, un cylindre ne peut qu'être approximé par des polygones, disons six rectangles pour la surface courbe et deux hexagones pour les faces terminales. Le nombre de polygones utilisé dans l'approximation détermine avec quelle précision l'objet est représenté et cela à une grande répercussion sur le temps pour modéliser l'objet, sur le coût de stockage, sur le temps pour le rendu et, bien entendu, sur la qualité de l'image générée. La popularité de ce modèle de représentation en synthèses d'images est certainement due au fait de sa simplicité inhérente.

Pour l'instant, nous considérons les objets comme des ensembles de sommets tridimensionnels et nous allons regarder comment ils sont transformés dans l'espace à trois dimensions en utilisant des transformations linéaires. Plus loin, nous nous intéresserons à savoir comment ces sommets sont connectés pour former des polygones et comment ces polygones sont structurés dans les objets complets (chapitres 2 et 3).

1.4 Les transformations affines

Les transformations géométriques affines sont des transformations qui opèrent sur les rotations, les redimensionnements, les translations et les déformations. Une transformation affine est construite à partir de n'importe quelle combinaison de transformations linéaires (rotation, redimensionnement et déformation), suivit par une translation qui n'est pas une transformation linéaire. Une transformation affine peut-être représentée par une matrice, et un ensemble de transformations affines peut être combiné pour former une seule transformation affine globale. Ces transformations ont la propriété de préserver le parallélisme des lignes, mais ne préservent pas les angles et les dimensions (figure 1.9).

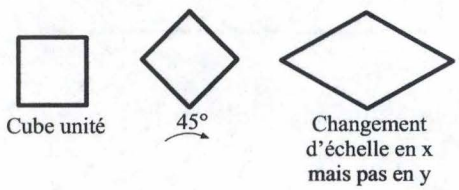


FIG. 1.9 – Un cube unité subit une rotation de 45° et un changement d'échelle non uniforme.

Pour définir géométriquement les objets d'une scène réelle, on utilise un repère cartésien[3]. Ce repère est constitué d'une origine O et de trois vecteurs normés linéairement

[3]la nature linéaire et orthogonale du repère cartésien a une conséquence réelle importante : ses propriétés le rendent semblable au monde où nous vivons.

indépendants notés $\vec{e_1}, \vec{e_2}, \vec{e_3}$[4]. Nous appellerons l'axe X, la droite passant par O dans la direction de $\vec{e_1}$; l'axe Y, la droite passant par O dans la direction $\vec{e_2}$ et l'axe Z, la droite passant par O dans la direction de $\vec{e_3}$.

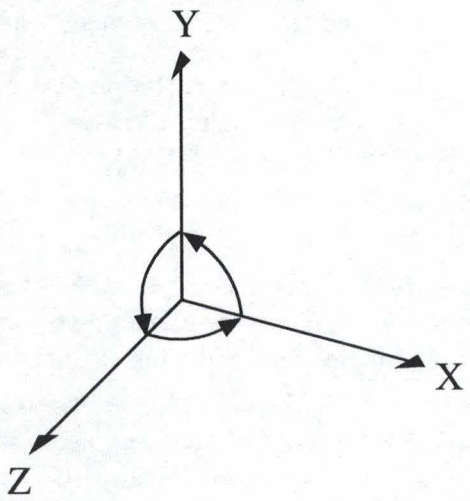


FIG. 1.10 – Repère droitier.

Les objets sont définis dans un repère de coordonnées qui est soit droitier ou gaucher. Le repère droitier est une convention standard en mathématique. La différence entre le repère droitier et le repère gaucher est juste le sens de l'axe des Z. Pour la suite, nous utiliserons toujours le repère droitier, c'est-à-dire $\vec{e_1} \times \vec{e_2} = \vec{e_3}$[5], qui est aussi une convention pour la librairie graphique OpenGL.

Il est très pratique de définir les objets d'une scène dans leur système de coordonnées local. Ce choix est judicieux pour trois raison :

1. lorsque l'on modélise un objet en trois dimensions, il est pratique de construire les sommets avec un point de référence dans l'objet. En fait, un objet complexe peut avoir beaucoup de systèmes de coordonnées locaux, un pour chaque sous-partie.
2. il peut arriver qu'un même objet apparaisse plusieurs fois dans une scène. "Instancier" un objet en lui appliquant des translations, rotations et changements d'échelle peut être vu comme la transformation du système de coordonnées local de l'objet vers le système de coordonnées global (repère absolu).
3. lorsqu'un objet subit une rotation, il est plus facile d'effectuer cette rotation si celle-ci est définie en respect avec un point de référence local tel que l'axe de symétrie.

[4] $\vec{e_i}$ représente le i^{eme} vecteur de la base canonique de $\mathbb{R}^3$

[5] l'opération notée "$\times$" est en fait l'opération de produit vectoriel

Un ensemble de sommets, appartenant à la représentation d'un objet, peut être transformés en un autre ensemble par des transformations. Chaque ensemble de sommets restent dans le même système de coordonnées.

En utilisant une notation matricielle, un point P, représenté par un vecteur colonne, est transformé en un point P' par une translation, changement d'échelle et une rotation comme suit :

$$\begin{cases} P' = P + d & \text{(translation)} \\ P' = SP & \text{(chg. d'échelle)} \\ P' = RP & \text{(rotation)} \end{cases}$$

où d est un vecteur de translation, S est une matrice diagonale de changement d'échelle et R est une matrice orthogonale de rotation.

Malheureusement, la translation du point P est traitée différemment de la rotation ou du changement d'échelle. Nous voudrions être capable de traiter ces trois transformations de la même façon et ainsi nous pourrions les combiner facilement.

Si les points sont exprimés en *coordonnées homogènes*, les trois transformations peuvent être traitées de manière identique. A cette fin, nous augmentons la dimension de l'espace, ainsi la translation devient une transformation linéaire. Un sommet V de coordonnées (x, y, z) est représenté par un qradruplet (X, Y, Z, w) où w est un facteur d'échelle différent de zéro. Il faut noter que deux coordonnées homogènes (X, Y, Z, w) et (X', Y', Z', w') représentent le même point si et seulement si une est un multiple de l'autre. Les conséquences sont qu'un point tridimensionnel a une multitude de représentation sous forme homogène.

Les coordonnées cartésiennes d'un point homogène sont données par

$$\begin{cases} x = \frac{X}{w} \\ y = \frac{Y}{w} \\ z = \frac{Z}{w} \end{cases}$$

En informatique graphique, pour passé outre du problème des multiples représentations, w est toujours fixé à 1. Les points dont la coordonnée $w = 0$ sont appelés *points à l'infini.*

Il existe une représentation géométrique des coordonnées homogènes. Chaque point dans le 3-espace (espace à trois dimensions) est représenté par une droite passant par l'origine dans le 4-espace, et la représentation homogénéisée de ces point forme un sous-espace tridimensionnel du 4-espace qui est défini par la simple équation $w = 1$.

Voyons à présent comment s'écrivent les matrices des transformations linéraires (rotations, etc.) adaptées aux coordonnées homogènes.

1.4.1 Translation

Soit

$$\mathrm{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

ainsi un point P de coordonnées homogènes $(x, y, z, 1)$ est transformé en un point P' tel que

$$TP = P'$$

c'est-à-dire

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} (x, y, z, 1)^T = (x + t_x, y + t_y, z + t_z, 1)^T$$

1.4.2 Rotation

La rotation d'un point P de coordonnées homogènes $(x, y, z, 1)$ se fait par rapport à l'origine O, autour de chaque axe.

Rotation d'un angle θ autour de l'axe x

$$\mathrm{R_x}(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

ainsi $P' = R_x(\theta)P$

Rotation d'un angle θ autour de l'axe y

$$\mathrm{R_y}(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

ainsi $P' = R_y(\theta)P$

Rotation d'un angle θ autour de l'axe z

$$\mathrm{R_z}(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

ainsi $P' = R_z(\theta)P$

Les colonnes de la sous-matrice supérieur gauche 3x3 de $R_x(\theta)$, $R_y(\theta)$ et $R_z(\theta)$ sont des vecteurs unités mutuellement perpendiculaires et la sous-matrice a un déterminant égal à 1, cela signifie que les trois matrices sont orthogonales. Nous pouvons rappeler que les transformations orthogonales préservent les distances et les angles.

1.4.3 Changement d'échelle

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

ainsi $P' = SP$

Toutes ces matrices de transformations ont des inverses. Nous obtenons l'inverse de la matrice de translation T en prenant la négation de t_x, t_y et t_z. Pour la matrice de changement d'échelle, il suffit de remplacer s_x, s_y, et s_z par leur valeur réciproque. En ce qui concerne les matrices de rotation, il suffit de prendre la négation de l'angle de rotation θ. Nous pouvons également utiliser la propriété de l'inverse d'une matrice orthogonale B. Cet inverse est juste la transposée de B, c'est-à-dire $B^{-1} = B^T$.

N'importe quelle combinaison de rotations, translations, changements d'échelle peut être multipliée ou concaténée pour donner une *matrice de transformation unique.*

Par exemple, si nous avons $P' = M_1 P$ et $P'' = M_2 P'$ alors les matrices de transformations M_1 et M_2 peuvent être concaténées pour donner

$$M_3 = M_1 M_2$$

et ainsi

$$P'' = M_3 P$$

Cette matrice de transformation unique sera toujours de la forme

$$M = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

1.5 Le système de visualisation

Bien que divers dispositifs d'affichage existent, la plupart des surfaces d'affichage graphiques des ordinateurs sont bidimensionnelles. C'est pourquoi, il est nécessaire d'utiliser divers procédés pour la conversion d'un espace de coordonnées tridimensionnelles vers une représentation bidimensionnelle. Ceux-ci doivent contenir une projection et une transformation de visualisation (changement de repère, etc.), le minimum requis pour la conversion d'une scène tridimensionnelle vers une projection en deux dimensions.

Nous allons donc créer une caméra virtuelle (figure 1.11) pouvant être située où nous désirons et orienté dans n'importe quelle direction. Cette caméra possède une position C_{pos} dans le *repère absolu*, une cible C_{cible} c'est-à-dire le point visé par la caméra, un

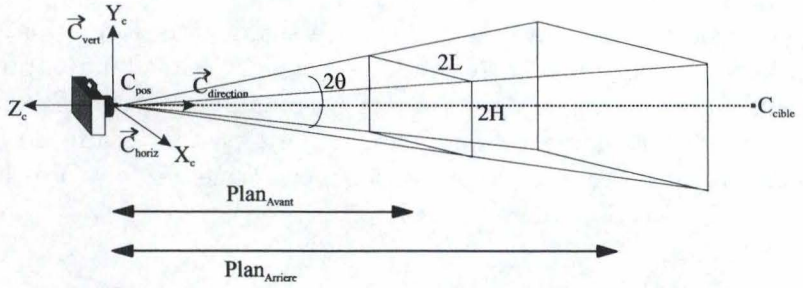


FIG. 1.11 – Caméra virtuelle

vecteur horizontal $\vec{C}_{horiz}$ et vertical $\vec{C}_{vert}$ déterminant son orientation, un angle d'ouverture $\frac{\theta}{2}$ déterminant le champ de vision et un plan de projection ($\simeq$écran) caractérisé par H et L, sa demi-hauteur et demi-largeur respectivement (Par la suite, nous considérerons que $H = L$).

Les trois premiers points ci-dessus permettent de définir le *repère de la caméra*, son origine est la position de la caméra dans le repère absolu. L'axe Z_c est dans la direction inverse du vecteur pointant vers le point cible, n'oublions pas que nous travaillons avec des repères droitiers, et les axes X_c, Y_c sont déterminés respectivement par les vecteurs horizontal et vertical.

Le processus de conversion d'un ensemble de points tridimensionnels, représentant un objet, vers une surface de visualisation bidimensionnelle se fait, comme on vient de le dire, en deux étapes.

La première étape est une transformation, notée T_{camera}, qui est un changement de repère convertissant les coordonnées des points de la scène vers le repère de la caméra. La seconde transformation, notée T_{proj}, projette les points tridimensionnels de la scène (qui sont dans le repère de la caméra) sur un plan de visualisation, appelé aussi écran. Cette étape élimine une dimension. La séparation des transformations de cette façon signifie que nous pouvons isoler la géométrie des projections du fait que, en général, le plan de projection peut avoir n'importe quelle position et orientation dans le repère absolu.

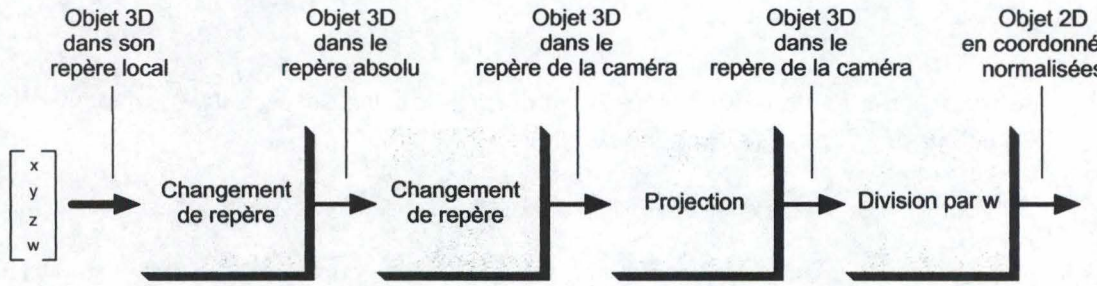


FIG. 1.12 – Processus de transformation.

Avant d'entrer en détail dans les étapes du processus de transformation, notons qu'il est possible d'optimiser quelque peu notre caméra virtuelle. Ceci en lui ajoutant un volume de visibilité appelé champ de vision de la caméra ou pyramide de vue. Si une portion de la scène se trouve en dehors de ce volume, alors il n'est pas nécessaire de lui appliquer les calculs de changement de repère, de projections, etc. Nous verrons plus loin en détail comment cela est réalisé (section 1.5.4 de ce chapitre).

1.5.1 Calculs préliminaires

Commençons tout d'abord par calculer quelques points importants. Connaissant l'angle d'ouverture θ et la dimension de l'écran $L = H$, nous allons pouvoir déterminer la distance, notée d, qui sépare l'écran de la position de la caméra. En effet,

$$d \tan(\theta) = L$$

donc,

$$d = \frac{L}{\tan(\theta)}$$

Ensuite, il va falloir calculer les vecteurs $\vec{C}_{vert}$, $\vec{C}_{horiz}$ et $\vec{C}_{direction}$ en fonction des données connues. Le vecteur unité donnant la direction dans laquelle est dirigée la caméra, noté $\vec{C}_{direction}$, est donné par

$$\vec{C}_{direction} = \frac{C_{cible} - C_{pos}}{\|C_{cible} - C_{pos}\|}$$

A partir de là, nous allons calculer le vecteurs $\vec{C}_{horiz}$ et $\vec{C}_{vert}$. Pour cela, nous calculons l'angle α (figure 5.2) qui est l'angle entre la projection du vecteur $\vec{C}_{direction}$ sur le plan $X_a Z_a$ et le vecteur $(0, 0, -1)$.

$$\begin{cases} \cos(\alpha) & = \frac{-z_{direction}}{\sqrt{x^2_{direction} + z^2_{direction}}} \\ \sin(\alpha) & = \frac{x_{direction}}{\sqrt{x^2_{direction} + z^2_{direction}}} \end{cases}$$

Le vecteur $\vec{C}_{horiz}$ est le vecteur $(1, 0, 0)$ qui à tourné autour de l'axe Y_a d'un angle de α dans le plan $X_a Y_a$,

$$\vec{C}_{horiz} = (\cos(\alpha), 0, -\sin(\alpha), 1)$$

Comme nous sommes dans un repère orthonormé droitier, le vecteur $\vec{C}_{vert}$ peut être déduit des vecteurs $\vec{C}_{horiz}$ et $\vec{C}_{direction}$. En effet,

$$\vec{C}_{vert} = -(\vec{C}_{direction} \times \vec{C}_{horiz}) = \vec{C}_{horiz} \times \vec{C}_{direction}$$

Si le vecteur $\vec{C}_{direction}$ est presque vertical, c'est-à-dire les composantes x et z sont très proches de zéro, alors le calcul de l'angle α comme nous venons de le faire est indéterminé.

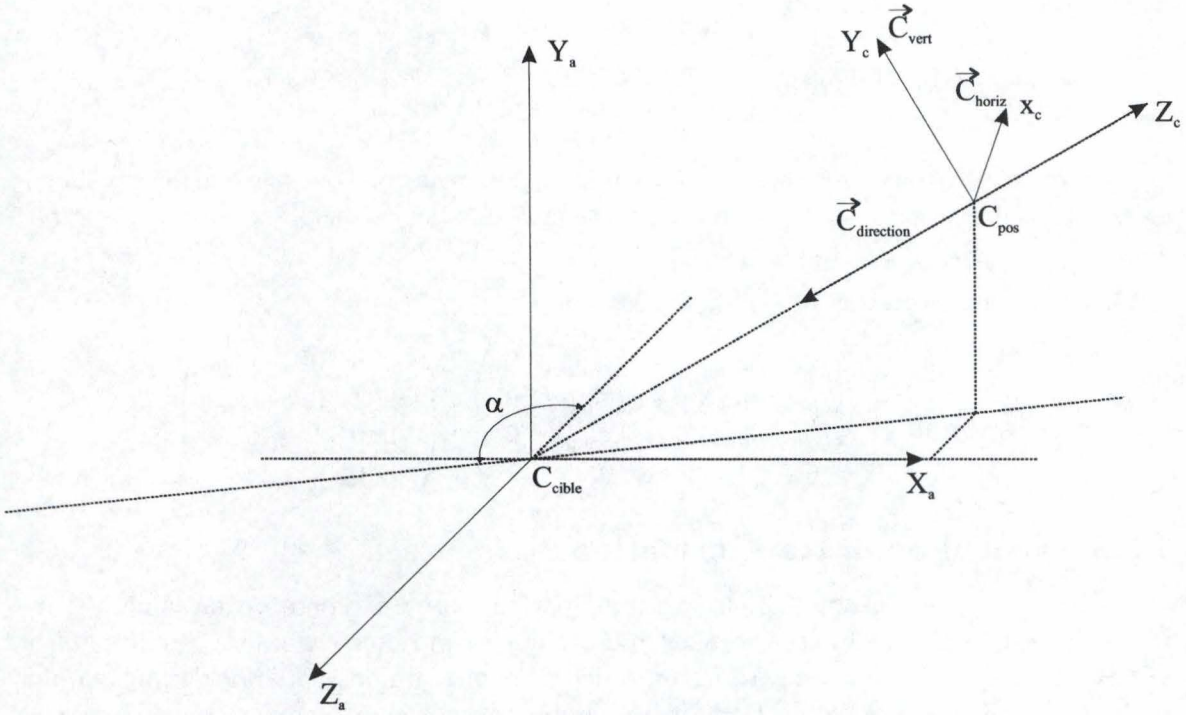


FIG. 1.13 – L'angle α est l'angle entre la projection du vecteur $\vec{C}_{direction}$ sur le plan X_aZ_a et le vecteur $(0, 0, -1)$.

1.5.2 Calcul de la transformation T_{camera}

Le but ici est de transformer les coordonnées d'un point du repère absolu vers le repère de la caméra, c'est-à-dire

$$(x_c, y_c, z_c, 1)^T = T_{camera}(x_a, y_a, z_a, 1)^T$$

Regardons de plus près comment s'effectue un changement de repère. Soient $(O_a, \vec{e_1}, \vec{e_2}, \vec{e_3})$ le repère absolu et $(O_c, \vec{f_1}, \vec{f_2}, \vec{f_3})$ le repère de la caméra. La formule de changement de repère est

$$(x_a, y_a, z_a, 1)^T = TR(x_c, y_c, z_c, 1)^T$$

où T est la matrice 4x4 représentant la translation de l'origine, c'est-à-dire $O_c - O_a$, et R est la matrice de rotation.

Ce qui nous intéresse, c'est la transformation inverse

$$(x_c, y_c, z_c, 1)^T = R^{-1}T^{-1}(x_a, y_a, z_a, 1)^T$$

Ici, la matrice R est égale à $\begin{pmatrix} x_{horiz} & x_{vert} & -x_{direction} & 0 \\ y_{horiz} & y_{vert} & -y_{direction} & 0 \\ z_{horiz} & z_{vert} & -z_{direction} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ [6]

Comme nous avons vu au point 1.4, les matrices de rotation sont des matrices orthogonales ayant comme caractéristique que l'inverse est égal à leur transposée. Donc $R^{-1} = R^T$, de plus, $O_a = (0,0,0,1)^T$ et $O_c = C_{pos}$.

Ainsi la transformation finale T_{camera} vaut

$$T_{camera} = \begin{pmatrix} x_{horiz} & x_{vert} & -x_{direction} & 0 \\ y_{horiz} & y_{vert} & -y_{direction} & 0 \\ z_{horiz} & z_{vert} & -z_{direction} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^T \begin{pmatrix} 1 & 0 & 0 & -x_{pos} \\ 0 & 1 & 0 & -y_{pos} \\ 0 & 0 & 1 & -z_{pos} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

1.5.3 Calcul de la transformation T_{proj}

Maintenant que les points sont exprimés par rapport au repère de la caméra, il ne reste plus qu'à calculer la transformation T_{proj} qui va projeter ces points sur le plan de visualisation. Lorsque nous regardons un objet s'éloigner, il nous apparaît de plus en plus petit. C'est ce qu'on appelle l'effet de *perspective*. Nous allons simuler cet effet avec la transformation T_{proj} ci-dessous.

Soit $(x_c, y_c, z_c, 1)$ les coordonnées d'un point exprimé dans le repère de la caméra.

Nous avons, par les formules des triangles semblables (figure 1.14) :

$$\frac{x_p}{d} = \frac{x_c}{z_c} ==> x_p = \frac{x_c}{\frac{z_c}{d}}$$

$$\frac{y_p}{d} = \frac{y_c}{z_c} ==> y_p = \frac{y_c}{\frac{z_c}{d}}$$

Nous pouvons en déduire

$$T_{proj} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{pmatrix}$$

---

[6] Considérons, par exemple, la matrice de rotation $R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$. Dans la sous-matrice supérieur gauche 2x2, considérons chacune des colonnes comme des vecteurs. Ces vecteurs présentent les trois propriétés suivantes :

- chacun d'eux est un vecteur unité ;
- chacun d'eux est perpendiculaire à l'autre ;
- le déterminant de la sous-matrice vaut 1.

Ces deux vecteurs sont en réalité deux vecteurs unités, pris respectivement le long des axes X et Y, ayant subis une rotation. Ces propriétés nous fournissent une manière de s'en sortir pour trouver une matrice de rotation lorsque l'on connaît l'effet désiré de la rotation.

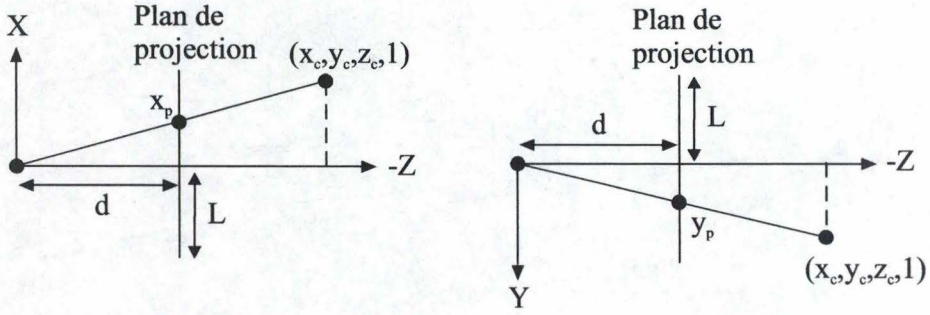


FIG. 1.14 – Projection du point $(x_c, y_c, z_c, 1)$ sur le plan de projection

ainsi

$$(x_p, y_p, z_p, w)^T = T_{proj}(x_c, y_c, z_c, 1)^T$$

Après correction, nous obtenons les coordonnées du point projeté

$$\begin{cases} x_p &= x_c/w \\ y_p &= y_c/w \end{cases} \quad \text{où} \quad w = \frac{z_c}{d}$$

Avant d'effectuer la projection des points, il serait bon de se poser la question : "les points sont-ils visibles par la caméra ?".

1.5.4 Visibilité d'un objet

Ce qui serai intéressant, pour économiser des opérations arithmétiques fort coûteuses en temps, c'est de pourvoir prédire si un objet de la scène est visible par la caméra. Nous allons tout d'abord définir un plan avant et un plan arrière perpendiculaire à l'axe Z_c délimitant ainsi le volume de visibilité (figure 1.15). Les composantes z des points où ces plans coupent l'axe Z_c sont notés respectivement $Plan_{avant}$ et $Plan_{arriere}$

Plaçons nous d'abord au niveau des points constituant l'objet. Soit P_c un point de l'objet exprimé dans le repère de la caméra.

Le premier critère pour décidé de la visibilité du point P_c est de vérifié si sa composante z_c est plus grande que le PlanAvant et plus petite que le PlanArriere.

Le deuxième critère est de vérifier que, étant donné l'angle d'ouverture θ, le point P_c se situe dans la pyramide de vue, c'est-à-dire, il faut que

1. l'angle α_1 entre l'axe Z_c et la droite passant par $(0, 0, 0, 1)$ et $(x_c, 0, z_c, 1)$ soit inférieur à l'angle d'ouverture θ, donc $\cos(\alpha_1) = \frac{-z_c}{\|(x_c,0,z_c,1)\|} > \cos(\theta)$.
2. l'angle α_2 entre l'axe Z_c et la droite passant par $(0, 0, 0, 1)$ et $(0, y_c, z_c, 1)$ soit inférieur à l'angle d'ouverture θ, donc $\cos(\alpha_2) = \frac{-z_c}{\|(0,y_c,z_c,1)\|} > \cos(\theta)$.

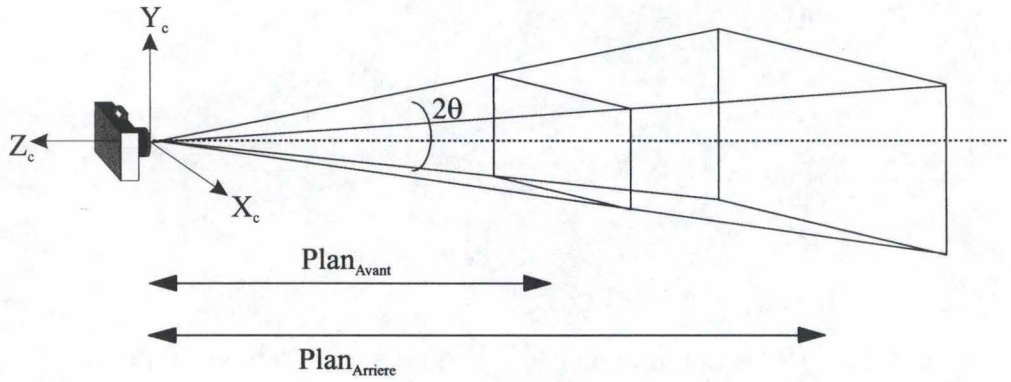


FIG. 1.15 – Le volume de visibilité est défini par l'angle d'ouverture θ, le plan avant et le plan arrière.

En pratique, nous utilisons pour ce dernier critère les tests suivants, plus rapides,

$$z_c^2 > \cos^2(\theta)(x_c^2 + z_c^2)$$
$$z_c^2 > \cos^2(\theta)(y_c^2 + z_c^2)$$

Plaçons nous maintenant au niveau de l'objet tout entier. En effet, à titre d'exemple, imaginons une scène modeste composée de 10 objets, constitués chacun de 1000 polygones, nous supposons ici que ce sont des triangles. S'il fallait à chaque mouvement de caméra vérifier si chaque point est visible, soit 30000 points, en plus de tout les autres traitements additionnels (gestion de la lumière, textures, etc.) nécessaire pour un rendu convenable, nous ne serions plus capable d'obtenir nos 20 images par seconde pour l'animation.

Donc, plutôt que de vérifier si chaque point est visible, supposons que l'objet est contenu dans une sphère et regardons si cette sphère est visible ou partiellement visible. Pour notre exemple ci-dessus, nous n'aurions que 10 tests à effectuer.

Pour ce faire, nous allons vérifier deux choses :

1. le centre de la sphère se trouve-t-il dans la pyramide ? Nous faisons cela facilement avec la méthode précédente qui vérifie si un point est visible. Si le centre est visible, alors la sphère l'est.
2. la sphère est-elle en intersection avec un des plans passant par les six faces de la pyramide tronquée ? Nous faisons cela en regardant si la distance entre le centre de la sphère et un des plans est plus petite que le rayon de la sphère.

Calculons les équations des plans passant par les six faces de la pyramide tronquée.

L'équation paramétrique du plan π_1 (figure 1.16) passant par les points de coordonnées homogènes $(0,0,0,1)$, $(\tan(\theta),\tan(\theta),0,1)$ et $(-\tan(\theta),\tan(\theta),0,1)$ est

$$\pi_1 \equiv \begin{cases} x &= 0 + \alpha - \tan(\theta) & +\beta\tan(\theta) \\ y &= 0 + \alpha\tan(\theta) & +\beta\tan(\theta) \\ z &= 0 + \alpha & +\beta \end{cases}$$

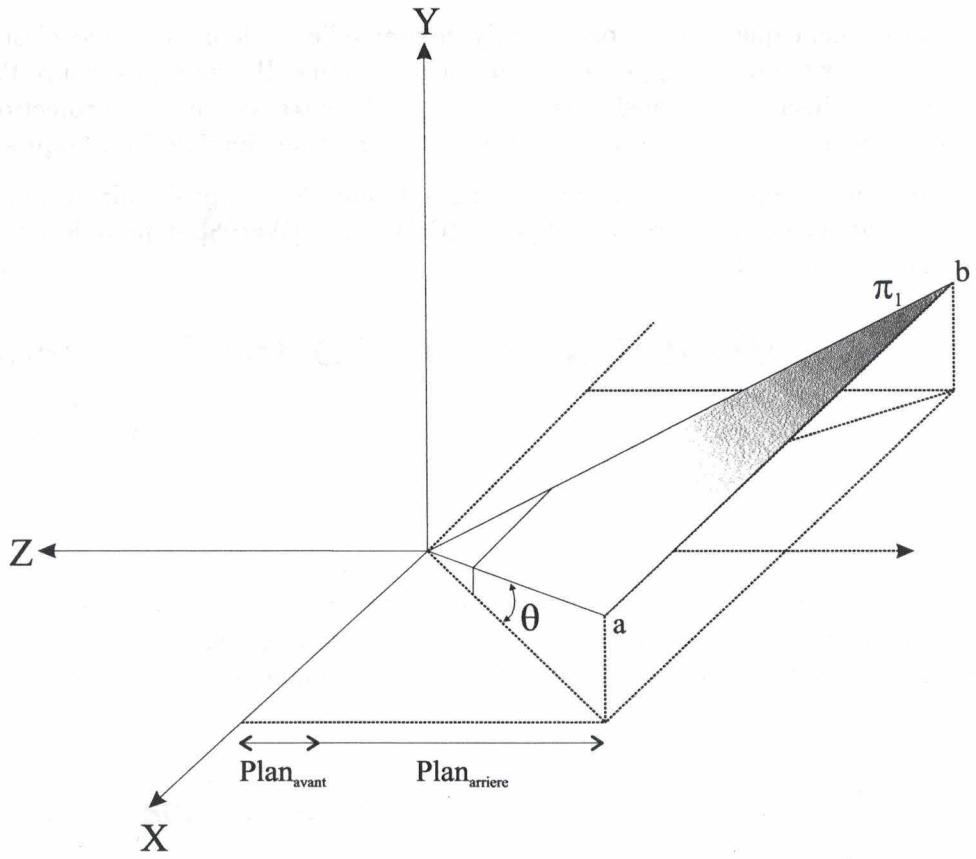


FIG. 1.16 – Le plan π_1 contient une des faces de la pyramide de vue tronquée

ce qui donne comme équation cartésienne

$$\pi_1 \equiv \begin{vmatrix} x-0 & -\tan\theta - 0 & \tan\theta - 0 \\ y-0 & \tan\theta - 0 & \tan\theta - 0 \\ z-0 & 1-0 & 1-0 \end{vmatrix} = 0$$

$$\iff 2y\tan\theta - 2z\left(\tan\theta\right)^2 = 0$$

Les cinq autres plans ont pour équation :

$$\begin{aligned} \pi_2 &\equiv 2y\tan\theta + 2z\left(\tan\theta\right)^2 = 0 \\ \pi_3 &\equiv 2z\left(\tan\theta\right)^2 - 2x\tan\theta = 0 \\ \pi_4 &\equiv -2x\tan\theta - 2z\left(\tan\theta\right)^2 = 0 \\ \pi_5 &\equiv z = Plan_{avant} \\ \pi_6 &\equiv z = Plan_{arriere} \end{aligned}$$

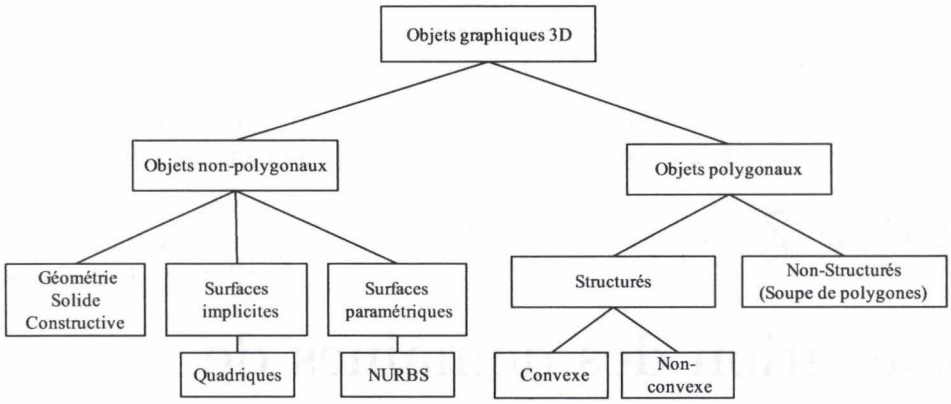


FIG. 2.1 – Taxinomie des représentations d'objets 3D.

objet, a été conçue pour faciliter le processus de modélisation des objets et le rendre plus intuitif. Cependant, il est fort peu adapté au problème de la détection de collision. En effet, pour obtenir la frontière d'un objet ainsi représenté, utile pour la détection, il faut passer par une phase difficile de transformation fort coûteuse en temps.

Dans la suite de ce mémoire, nous ne considérerons plus que les représentations par frontière qui sont les plus adaptées pour notre problème. En effet, elles peuvent facilement contenir des informations topologiques sur la géométrie d'un objet telles que l'adjacence ou l'incidence entre composants[1] (i.e., sommets, arêtes, polygones) de l'objet ainsi que leur orientation ; elles sont très simple à gérer. Elles permettent surtout aux algorithmes de détection de collision qui les utilisent d'atteindre des performances temps-réels, chose impossible avec la CSG.

Les objets graphiques peuvent être représentés de différentes manières avec une représentation par frontière (polygonale). Nous pouvons citer :

- les collections de polygones ou "soupe" de polygones : rien n'est exigé sauf que les polygones soient plans. Les objets représentés ainsi n'ont même pas besoin d'être fermés.
- les polyèdres (fermés) : un intérieur et un extérieur peuvent être définis et exploités.
- les polyèdres constitués uniquement de polygones convexes, les polyèdres peuvent être non-convexes.
- les polyèdres convexes : cette classe est certainement la plus raisonnable. Elle semble fournir de grands avantages pour les algorithmes incrémentaux.

[1] voir l'annexe A page 95 pour obtenir la terminologie anglaise.

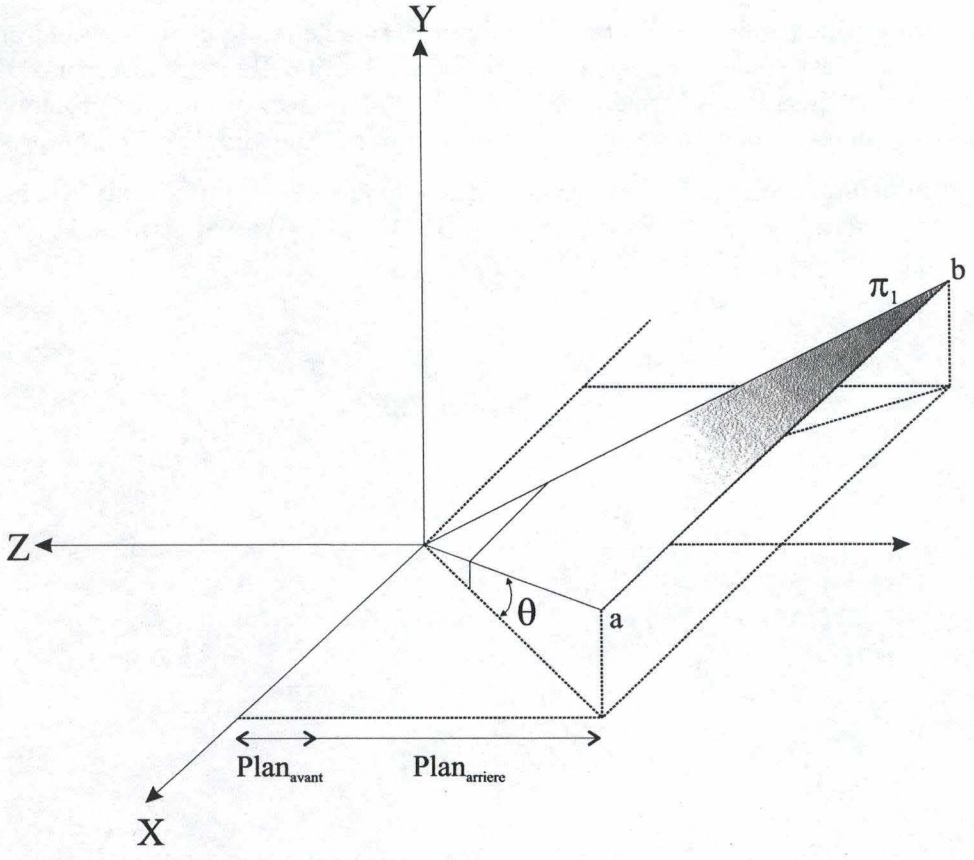


FIG. 1.16 – Le plan π_1 contient une des faces de la pyramide de vue tronquée

ce qui donne comme équation cartésienne

$$\pi_1 \equiv \begin{vmatrix} x-0 & -\tan\theta - 0 & \tan\theta - 0 \\ y-0 & \tan\theta - 0 & \tan\theta - 0 \\ z-0 & 1-0 & 1-0 \end{vmatrix} = 0$$

$$\iff 2y\tan\theta - 2z\left(\tan\theta\right)^2 = 0$$

Les cinq autres plans ont pour équation :

$$\begin{aligned} \pi_2 &\equiv 2y\tan\theta + 2z\left(\tan\theta\right)^2 = 0 \\ \pi_3 &\equiv 2z\left(\tan\theta\right)^2 - 2x\tan\theta = 0 \\ \pi_4 &\equiv -2x\tan\theta - 2z\left(\tan\theta\right)^2 = 0 \\ \pi_5 &\equiv z = Plan_{avant} \\ \pi_6 &\equiv z = Plan_{arriere} \end{aligned}$$

La caméra virtuelle que nous venons d'analyser simule l'effet de perspective, c'est pourquoi cette caméra est souvent appelée *caméra en perspective*. Il existe beaucoup d'autres types de caméra utilisant des transformations T_{proj} différentes, comme les projections orthographiques (nous aurons l'occasion d'en toucher un mot au chapitre 5), obliques, etc.

Ceci termine notre aperçu de l'informatique graphique. Pour approfondir le sujet, nous pouvons nous référencer aux livres [FvDFH93], [FvDF+97], [Wat98] et pour la géométrie algorithmique au livre [O'R98].

Chapitre 2

Classification des domaines de problèmes

Nous allons faire dans ce chapitre une classification des différentes questions qu'il faut se poser avant d'entreprendre l'analyse proprement dite du problème de la détection de collision. Notamment, la question du choix de la représentation des objets, la question de ce qu'il est possible de demander à un algorithme de détection de collision et d'attendre de lui, etc.

2.1 Représentation des objets

Il existe beaucoup de types de représentation d'objet utilisés en CAO-PAO et en graphisme 3D. La représentation interne d'objets graphiques a une grande importance sur le choix des algorithmes, pas seulement pour la détection de collision, mais aussi pour le rendu, la modélisation et beaucoup d'autres parties d'un système graphique interactif.

Durant ces trente dernières années, plusieurs approches différentes pour représenter un objet ont émergé. Une classification très simple de ces différentes approches peut être faite :

- les représentations *basées sur la frontière* sont la représentation classique par frontière (b-rep) également appelée représentation par treillis de polygone (polygon mesh), les surfaces libres de forme et les représentations par frontières hiérarchiques ;
- les représentations *basées sur le volume* sont les arbres octaux, le partitionnement binaire de l'espace et la géométrie solide constructive.

Une taxinomie des représentations d'objets tridimensionnels peut être faite, sur base de la classification ci-dessus, en séparant bien les représentations polygonales des non-polygonales (figure 2.1).

Toutes ces représentations d'objets ont été imaginées pour répondre à des besoins spéciaux. Par exemple, la géométrie solide constructive (CSG), dans laquelle on dispose de primitives volumiques paramétrées (cubes, sphères, cônes, etc.), de transformations géométriques et d'opérations booléennes (union, intersection, différence) pour modéliser un

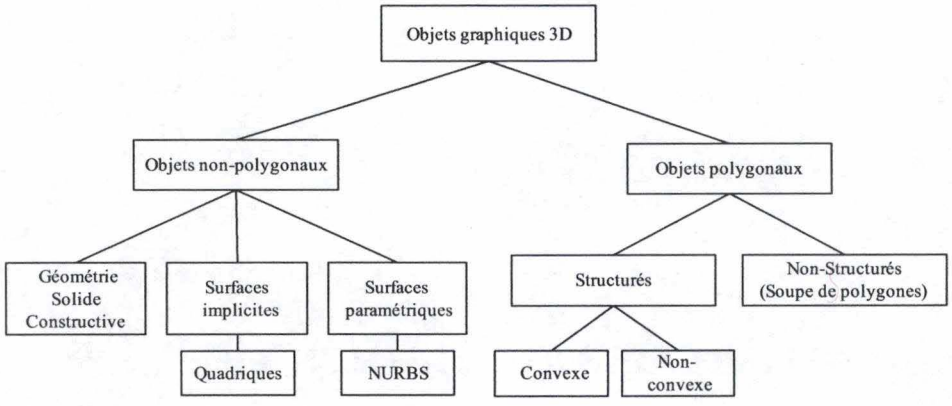


FIG. 2.1 – Taxinomie des représentations d'objets 3D.

objet, a été conçue pour faciliter le processus de modélisation des objets et le rendre plus intuitif. Cependant, il est fort peu adapté au problème de la détection de collision. En effet, pour obtenir la frontière d'un objet ainsi représenté, utile pour la détection, il faut passer par une phase difficile de transformation fort coûteuse en temps.

Dans la suite de ce mémoire, nous ne considérerons plus que les représentations par frontière qui sont les plus adaptées pour notre problème. En effet, elles peuvent facilement contenir des informations topologiques sur la géométrie d'un objet telles que l'adjacence ou l'incidence entre composants[1] (i.e., sommets, arêtes, polygones) de l'objet ainsi que leur orientation ; elles sont très simple à gérer. Elles permettent surtout aux algorithmes de détection de collision qui les utilisent d'atteindre des performances temps-réels, chose impossible avec la CSG.

Les objets graphiques peuvent être représentés de différentes manières avec une représentation par frontière (polygonale). Nous pouvons citer :

- les collections de polygones ou "soupe" de polygones : rien n'est exigé sauf que les polygones soient plans. Les objets représentés ainsi n'ont même pas besoin d'être fermés.
- les polyèdres (fermés) : un intérieur et un extérieur peuvent être définis et exploités.
- les polyèdres constitués uniquement de polygones convexes, les polyèdres peuvent être non-convexes.
- les polyèdres convexes : cette classe est certainement la plus raisonnable. Elle semble fournir de grands avantages pour les algorithmes incrémentaux.

[1] voir l'annexe A page 95 pour obtenir la terminologie anglaise.

2.2 Différents types de demandes

Dans le cas le plus simple, nous voudrions savoir si deux objets se touchent. Parfois, il serait intéressant de savoir quelles parties se touchent, c'est-à-dire trouver leurs intersections. D'autres fois encore, nous voudrions connaître leur séparation, si deux objets sont disjoints alors quelle est la distance Euclidienne minimale les séparant ? Par contre, s'ils s'interfèrent alors quelle est la distance translationnelle minimale requise pour les séparer ? Finalement, si nous connaissons la position et la vitesse des objets, nous pourrions nous demander quand aura lieu la prochaine collision ? C'est le calcul de l'ETA ou temps estimé d'arrivée.

Des applications différentes requièrent des demandes différentes. L'information sur la distance entre deux objets est utile pour le calcul les forces d'interaction et les fonctions de pénalité pour les simulations dynamiques ou le planning des déplacements d'un robot. Le calcul de l'intersection est très important pour les simulations physiques et systèmes d'animation qui doivent connaître toutes les zones de contact afin de calculer la réponse aux collisions.

Au vu de ces demandes, nous pouvons déjà nous attendre à plusieurs algorithmes de détection de collision, des plus rudimentaires aux plus sophistiqués suivant les besoins.

2.3 Environnements de simulation

Les caractéristiques d'une simulation sont souvent utilisées pour la conception et le choix de l'algorithme de détection de collision le plus approprié. Voici quelques cas que nous allons analyser :

- traitement par paire d'objets contre traitement multi-objets :
  Si le problème n'implique qu'une paire d'objets alors nous donnons à l'algorithme la caractéristique de "traitement par paire". Si par contre nous avons plusieurs objets différents à traiter, alors nous le caractérisons de "traitement multi-objets".
- mouvements : statique contre dynamique :
  Les demandes sont souvent exécutées à maintes reprises sur les mêmes objets dans un même environnement, pendant que les objets subissent des rotations et translations en des pas successifs de temps. Dans ces environnements dynamiques, les relations géométriques ne diffèrent que très légèrement de tranche de temps en tranche de temps lorsque le mouvement entre pas successifs de temps est relativement petit. Les algorithmes qui utilisent cette propriété exploitent en fait la propriété appelée de cohérence temporelle. Par exemple, une translation des objets à vitesse constante faible permet d'exploiter la cohérence temporelle.
  Parfois le problème implique des objets qui ne sont pas en mouvement. Par exemple, pour un groupe moteur dont les ingénieurs ont une représentation, il serait intéressant d'effectuer des contrôles d'interférence statiques parmi les composants pour vérifier la tolérance et l'accessibilité pour l'entretien.

- objets rigides contre objets déformables :
  Lorsque la composante du temps est introduite, il y a aussi la possibilité que les objets se déforment au cours du temps. En supposant que les déformations entre pas successifs de temps sont petites, certains algorithmes pourraient exploiter ici encore la propriété de cohérence temporelle.

2.4 Différents types de détection de collision

Les algorithmes de détection de collision peuvent être classifiés par plusieurs critères :

- *approximatifs contre exacts* : la détection de collision approximative est habituellement biaisée, c'est-à-dire que l'algorithme tend à favoriser une réponse plutôt qu'une autre. Ce biais est causé par l'utilisation de simplification géométrique ou d'algorithmes probabilistes (section 4.4). Par exemple, si nous disposons d'un algorithme de détection de collision très performant pour les objets convexes et que nous l'utilisons sur l'enveloppe convexe d'objets concaves, nous voyons tout de suite que cette utilisation est fort approximative.
- *utilisant le temps comme une quatrième dimension contre uniquement du tridimensionnel* (géométrie intemporelle) : la dimension du temps peut être utilisée pour calculer le moment exact d'une collision ou pour exploiter la propriété de cohérence temporelle en vue d'accélérer la procédure de détection. Les approches intemporelles considèrent les objets uniquement en un temps donné, mais gardent à l'esprit que les objets sont probablement en mouvement.
- *utilisant des hiérarchies d'objets ou non* : pour accélérer la détection lorsque l'objet entier est gigantesque (en nombre de polygones).
- *restreints à un domaine ou non* : le plus souvent, à la classe des polyèdres convexes. D'autres restrictions seraient aux objets fermés ou aux polyèdres constitués de polygones convexes.
- *utilisant des objets flexibles ou bien des objets rigides* : l'usage de la détection de collision avec des objets flexibles, qui peuvent changer leur géométrie au cours du temps, complique sensiblement le problème parce qu'aucun pré-calcul ne peut être effectué ou bien il doit être fait à chaque changement de géométrie.
- *travaillant en direct contre en différé* : un certain nombre d'applications peuvent s'en sortir avec une détection de collision "en différé", parce que l'application n'est pas pilotée par des entrées en temps réel comme dans les environnements de réalité virtuelle. Pour exemple, nous pouvons citer les simulations basées sur les phénomènes physiques pour l'animation.
- *incrémentaux contre faisant "table rase"* : les méthodes incrémentales essayent d'exploiter les résultats d'une demande antérieure. C'est une forme d'exploitation de la propriété de cohérence temporelle.

Une petite remarque peut être faite à propos du caractère approximatif d'un algorithme. Bien que dans les environnements de réalité virtuelle, il n'est pas nécessaire de connaître le point exact de collision, une détection trop imprécise trouble l'impression de réalisme (eg.

utiliser seulement des tests avec des boîtes limites).

2.5 Le temps réel non sans difficultés

Il existe plusieurs difficultés qui surviennent généralement lorsque le but est la détection en temps réel de collision :

- *la détection entre paires d'objets* (pairwise tests) : un algorithme naïf testerait toutes les paires possibles de faces et d'arêtes d'une paire d'objets et fera de même pour toutes les paires possibles d'objets de l'environnement virtuel. Ainsi si un environnement dispose de n objets, alors le nombre de tests serait de n^2.
- *la discrétisation du temps* : il devient alors très difficile de calculer le moment exact de la collision. En effet, les systèmes dynamiques graphiques affichent les objets à certains intervalles de temps, habituellement dès que l'application en a fini avec les calculs tels que rassembler les données d'entrée, déplacer les objets, etc. Si le module de détection de collision ne "voit" l'environnement seulement qu'à ces moments donnés sans aucune information sur le futur, alors il peut uniquement vérifier s'il y a une collision ou pas. Maintenant, si des vitesses (rotationnelles et translationnelles) et une accélération sont fournies comme attribut aux objets, alors le module sera en mesure de calculer le moment exact de la collision (ou une approximation de cela).Toutefois, cela ne pose plus un problème à partir du moment où les applications essayent de rendre ces intervalles aussi petit que possible.
- *l'utilisation d'objets non-convexes* : ou encore pire, des objets qui ne sont pas uniquement constitués de polygones convexes et qui ne sont pas fermés. Il existe un grand nombre de méthodes pour aborder le problème de la détection de collision entre objets convexes ; pour les objets non-convexes, très peut d'algorithmes sont connus (section 6.2). Pour des polyèdres non-convexes, nous avons toujours la possibilité d'utiliser les algorithmes prévus pour les convexes seulement si nous les partitionnons en morceaux convexes (algorithme NP-difficile). Par contre, si les objets ne sont pas fermés, alors il n'y a pas grand chose que nous pouvons faire.

2.6 Méthodes générales d'optimisation de la détection de collision

Il y a quelques caractéristiques qui peuvent être exploitées pour accélérer la détection, ceci aussi bien au niveau de la détection entre paire d'objets qu'au niveau global impliquant une multitude d'objets en mouvement :

- toutes sortes de *volumes limites*, pour les objets aussi bien que pour les polygones.
- *cohérence spatiale* utilise le fait qu'habituellement de grandes régions de l'espace ne sont occupées que par un seul objet ou rien du tout.
- *cohérence temporelle* exploite le fait que les objets en mouvement se déplacent habituellement sur une trajectoire continue.

- *pré-calcul*, en ajoutant aux données géométriques des structures de données additionnelles, la conception d'algorithmes plus efficaces devient possible. Le principal inconvénient de cette approche est que ce type d'algorithme ne peut être utilisé lorsque la géométrie des objets change souvent au cours du temps d'exécution.

2.7 Ce que nous attendons d'un algorithme de détection de collision

En général, il est très intéressant qu'un tel algorithme ait les qualités suivantes. Il doit être :

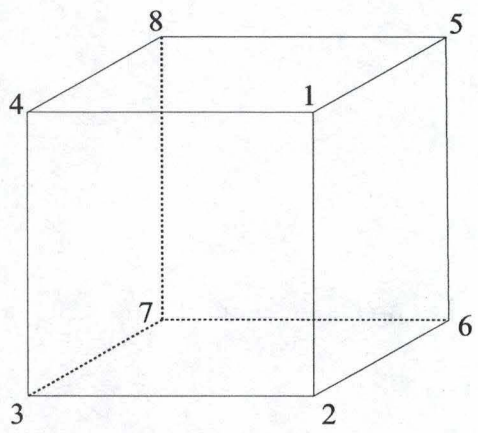
- rapide (si possible temps réel) ;
- approprié pour une classe d'objets graphiques aussi large que possible ;
- exact (c-à-d que le module de détection de collision signale une collision si et seulement si il y a une intersection de surfaces) ;
- capable de donner un "témoin" c'est-à-dire une arête et/ou polygone, où les deux objets sont en collision (si possible, rapporter tous les points de collision) ;
- capable de prendre en charge plusieurs objets en mouvement.

Chapitre 3

Les structures de données des représentations par frontière

3.1 Introduction

Dans beaucoup de cas, la très simple structure de données qui consiste en une liste des sommets et polygones suffit pour les algorithmes simples (figure 3.1). Mais dès que nous voulons apporter des perfectionnements à ces algorithmes (exploité la proximité, etc.), nous avons besoin d'une structure de données plus riche. Par exemple, vérifier si un objet n'est pas "2-diversifié"[1], c'est-à-dire qu'il existe plus de deux polygones incidents à la même arête, serait assez long si ne disposions que d'une simple liste des sommets et polygones associés.



Liste des sommets			
Sommet	*x*	*y*	*z*
1	1	1	1
2	1	-1	1
3	-1	-1	1
4	-1	1	1
5	1	1	-1
6	1	-1	-1
7	-1	-1	-1
8	-1	1	-1

Liste des faces				
Face	*sommet 1*	*sommet 2*	*sommet 3*	*sommet 4*
1	1	2	3	4
2	4	3	7	8
3	8	7	6	5
4	5	6	2	1
5	1	4	8	5
6	2	3	7	6

FIG. 3.1 – Un cube définit par ses huit sommets et ses six faces

Comme d'ordinaire en informatique, les structures de données et les algorithmes dé-

[1] "2-manifold" en anglais

pendent l'un de l'autre (parfois ils sont juste une manière différente d'aborder un problème). Des structures de données plus riches fournissent l'opportunité pour la création d'algorithmes plus efficaces.

3.2 Structures de données

La liste additionnelle habituellement désiré est une liste des arêtes. Beaucoup d'algorithmes présentés dans ce mémoire (entre deux objets) utilisent les arêtes et polygones comme les composants de base d'un polyèdre.

Ajouter une liste des arêtes permet de construire une structure de données contenant des informations sur l'incidence et l'adjacence. Nous allons voir ci-dessous quelques structures de données intéressantes pour la gestion des arêtes, nous avons déjà une liste des sommets et des polygones.

3.2.1 Winged-Edge

Cette structure a été développée par Baumgart il y a presque trente ans ([Bau72]). Une arête stocke ses deux sommets extrémités et ses deux polygones (faces) incidents, c'est-à-dire, pour lesquels elle est une frontière. En plus de cela, chaque arête mémorise des références vers quatre arêtes adjacentes. Et finalement, chaque sommet et chaque face stockent un pointeur arbitraire vers une de ses arêtes incidentes et frontières respectivement.

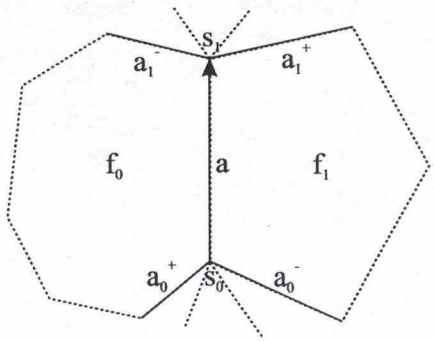


FIG. 3.2 – La structure de données "winged-edge". Les arêtes en pointillés ne sont pas stockées dans la structure

En résumé, l'enregistrement d'une arête a consiste en huit pointeurs :
- vers les deux extrémités de a : s_0 et s_1 ;
- vers les deux faces adjacentes à a : f_0 et f_1 qui sont respectivement à "gauche" et à "droite" de $s_0 s_1$;

- vers quatre arêtes (les ailes[2] de a) : a_0^- et a_0^+ les arêtes incidentes à s_0, dans le sens des aiguilles d'une montre et dans le sens inverse des aiguilles d'une montre autour de s_0 respectivement ; et a_1^- et a_1^+ les arêtes incidentes à s_1.

Voyons à présent un exemple d'utilisation. Les arêtes frontières d'une face f peuvent être trouvées en récupérant l'unique arête a stockée dans l'enregistrement de f, et il suffit de suivre les arêtes a^+ autour de f jusqu'à rencontrer à nouveau a. Cependant, puisque a est orienté arbitrairement, il est nécessaire de vérifier si f est à gauche ou à droite de a afin de décider s'il faut suivre a_1^+ ou a_0^-.

3.2.2 Quad-Edge

Au lieu de stocker les informations des deux cotés d'une arête, les arêtes sont stockées deux fois dans chaque direction, chaque direction stocke la moitié de l'information. Cela permet de parcourir la structure aussi bien dans le sens des aiguilles d'une montre que le sens contraire.

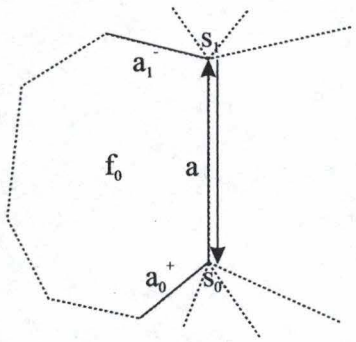


FIG. 3.3 – La structure de données "quad-edge". Les arêtes en pointillés ne sont pas stockées dans la structure

Cette structure de données est moins économique en mémoire que la structure "winged-edge".

3.2.3 Liste d'arête doublement chaînée

Cette structure de donnée est une version allégée de la structure "winged-edge" qui contient deux pointeurs vers des arêtes en moins (comme nous pouvons le voir sur la figure 3.4).

C'est certainement à cause de sa consommation de mémoire moins élevée que les deux autres structures de données qui rend cette structure fort appréciée. Elle est la plus souvent utilisée dans les algorithmes de détection de collision.

[2]the " wings " of a

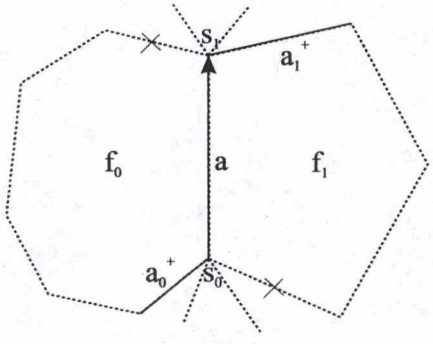


FIG. 3.4 – La structure de données constituée d'une liste d'arête doublement chaînée. Les arêtes en pointillés ne sont pas stockées dans la structure

Chapitre 4

La détection de collision entre paire d'objets (approche algorithmique)

4.1 Introduction

Dans ce chapitre, nous allons décrire quelques algorithmes de détection de collision entre une paire d'objets, c'est-à-dire qu'ils sont la solution du problème de décision suivant :

"Etant donné deux objets A et B, est-ce que A et B sont en intersection ?"

Comme nous avons vu au chapitre 2, il existe différentes type d'objets, ce qui implique différentes définitions du terme *collision*. Par exemple :

- *objets arbitraires* : nous dirons que deux objets arbitraires A et B sont en collision si et seulement si il existe une arête a de A et un polygone p de B tel que l'intersection de a avec p soit différente du vide.
- *objets fermés* : deux objets fermés A et B sont en collision si et seulement si il existe un point de l'espace x qui appartienne à la fois à A et B

  ou

  deux objets fermés A et B ne sont pas en collision si et seulement si $d(A, B) = \min\{|a - b| : a \in A, b \in B\} > 0$
- *objets convexes* : deux objets convexes A et B ne sont pas en collision si et seulement si il existe un plan π séparateur tel que tout élément de A se trouve d'un côté de π, et tout élément de B se trouve de l'autre côté de π, c'est-à-dire qu'aucun segment de droite joignant deux points de A ne coupe π et aucun segment de droite joignant deux points de B ne coupe π.

4.2 Les objets arbitraires

Nous allons traiter ici les objets constitués d'une collection de polygones plans. Un objet de ce type peut être en intersection avec lui-même, c'est-à-dire que des polygones le constituant peuvent être en intersection. Il est très important qu'un algorithme de détection de

collision puisse travailler avec ce type d'objet, puisque beaucoup de données géométriques, provenant de logiciel de CAO, sont généralement des objets mal formés.

L'algorithme que nous allons examiner est très simple. Il fonctionne de la manière suivante.

Soient A et B deux objets arbitraires. Nous pouvons sans perte de généralité, supposer que les polygones constituant les objets A et B sont des triangles. En effet, tout polygone peut être partitionné en triangles [O'R98]. Soient t_A^i $(i = 1, .., n)$ les n triangles constituant A et t_B^i (i=1,..,m) les m triangles constituant B. Nous supposons également, pour une question de facilité, que A et B sont exprimés dans le même système de coordonnées.

L'algorithme va vérifier pour chaque arête appartenant à A si elle est en intersection avec un polygone appartenant à B, et vice versa. En effet, il n'est pas suffisant de vérifier seulement si les arêtes de A sont en contact avec les polygones de B. Pour s'en convaincre, il suffit de regarder le contre-exemple suivant (figure 4.1).

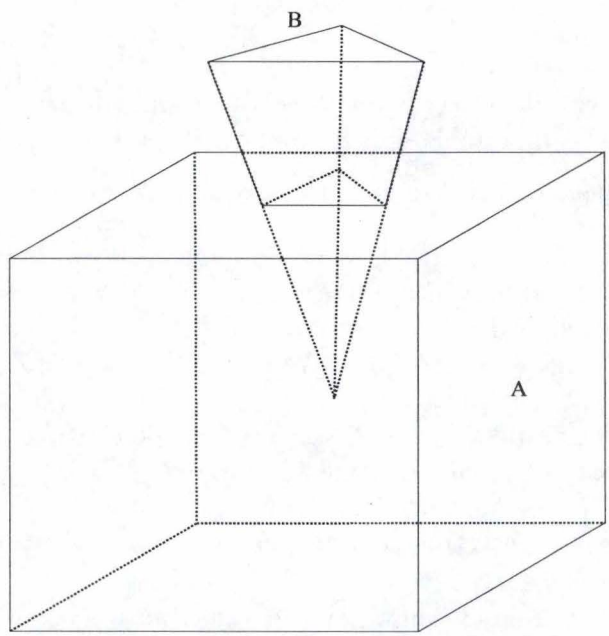


FIG. 4.1 – Contre-exemple : il n'est pas suffisant de vérifier seulement si les arêtes de A sont en contact avec les polygones de B

L'algorithme va devoir tester, à un moment donné, si une arête quelconque est en intersection avec un triangle.

Soit $a = qr$ une arête d'un triangle t_A^i $(i = 1, .., n)$ avec q et r deux sommets du triangle et b un triangle t_B^i $(i = 1, .., m)$ avec comme vecteur normal $\vec{v}$.

Pour réaliser cela, trois étapes sont nécessaires :

1. vérifier si la droite associée à l'arête a est parallèle au plan contenant le triangle b. Si ce n'est pas le cas, passer à l'étape suivante.
2. vérifier si le point d'intersection entre la droite et le plan est situé dans l'arête a. Si c'est le cas, alors passer à la dernière étape.
3. vérifier si le point trouver à l'étape 2 est dans le triangle b. Si c'est le cas, alors l'arête a est en intersection avec le triangle b.

4.2.1 1ère étape

Pour voir si l'arête a est parallèle au plan qui contient le triangle, projetons simplement le vecteur (r-q) sur le vecteur normal v et vérifions que cette projection est nulle, c'est-à-dire

$$v \cdot (r - q) = 0$$

4.2.2 2ème étape

Déterminons si l'arête reliant q et r est en intersection avec le plan π contenant le triangle b (figure 4.2). L'équation du plan π peut être vue comme le produit scalaire

$$\pi \equiv (x, y, z) \cdot v = d$$

où d est la distance de l'origine à π.

N'importe quel point de l'arête peut être représenté par $p(t) = q + t(r - q)$ où $t \in [0, 1]$. Calculons à présent la valeur du paramètre t qui va positionner p sur le plan π. Du fait que tout point du plan doit satisfaire l'équation ci-dessus, nous avons

$$p(t) \cdot v = d$$

$$[q + t(r - q)] \cdot v = d$$

$$t = \frac{d - q \cdot v}{(r - q) \cdot v}$$

La seule donnée manquante est d, mais elle est très facilement calculable en remplaçant (x, y, z) de l'équation du plan par un sommet quelconque du triangle b. Le résultat t va nous permettre de savoir si le point est sur l'arête, en effet, il suffit de vérifier si $t \in [0, 1]$.

4.2.3 3ème étape

Il nous reste à vérifier si le point p est à l'intérieur ou à l'extérieur du triangle b. Il existe beaucoup d'algorithmes permettant de résoudre ce problème. Nous allons analyser ici l'un des plus efficace basé sur le calcul d'aire et un autre, un peu moins efficace basé sur les angles.

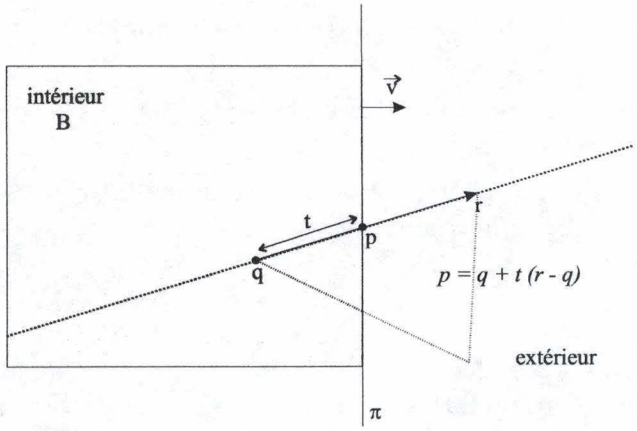


FIG. 4.2 – Intersection d'une arête qr avec un triangle $b \in B$ (coupe transversale)

Méthode basée sur l'aire

Soient e, f, g et T les trois sommets et l'aire du triangle b. Puisque nous sommes arrivés à la troisième étape, nous en concluons que e, f, g et p sont coplanaire ($\in \pi$). Si le point p se situe dans le triangle b, alors nous pouvons décomposer le triangle b en trois triangles :

- le triangle de sommets e, p, f et d'aire T_1 ;
- le triangle de sommets f, p, g et d'aire T_2 ;
- le triangle de sommets g, p, e et d'aire T_3 ;

et l'aire T est égale à la somme des trois aires T_1, T_2 et T_3.

Si le point p est extérieur au triangle b, alors $T < T_1 + T_2 + T_3$.

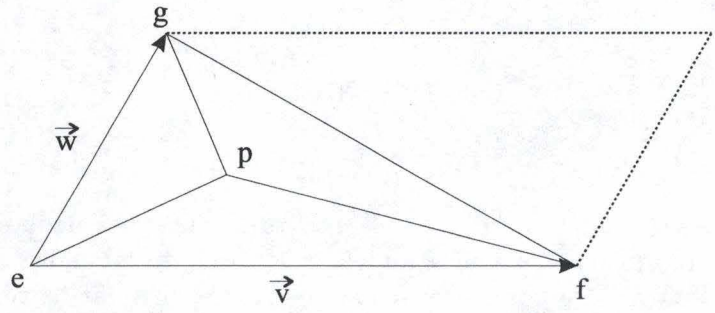


FIG. 4.3 – La norme du produit vectoriel des vecteurs $\vec{v}$ et $\vec{w}$ donne l'aire du parallélogramme

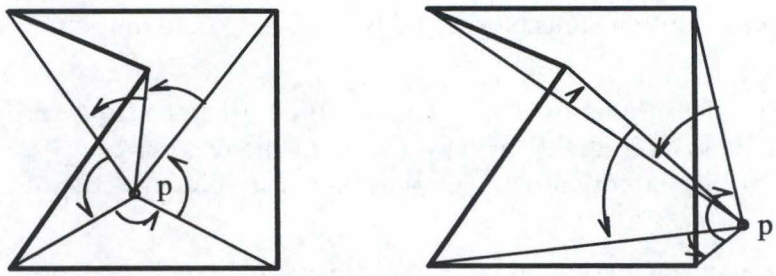
L'aire d'un triangle est facilement calculable si nous connaissons ces trois sommets. En effet, prenons l'exemple du triangle b (figure 4.3). Nous savons par une propriété du produit vectoriel que $\|\vec{v} \times \vec{w}\|$ = aire du parallélogramme. Donc, l'aire du triangle b est donnée par

la formule

$$T = \frac{\|(f-e) \times (g-e)\|}{2}$$

Dans une implémentation réelle, les erreurs d'imprécisions (arrondi, etc.) sont omniprésentes. En règle générale, nous considérerons que deux scalaires s_1 et s_2 sont égaux si $|s_1 - s_2| < \epsilon$ où ϵ est un seuil de tolérance à fixer (assez petit).

Méthode basée sur les angles



(a) la somme des angles vaut 2π (b) la somme des angles vaut 0

FIG. 4.4 – Le test point-dans-polygone avec la méthode de la somme des angles

Soient e, f, g les trois sommets du triangle b. Si le point p se situe dans le triangle b, alors la somme des angles $\measuredangle((e-p),(f-p))$, $\measuredangle((f-p),(g-p))$ et $\measuredangle((g-p),(e-p))$ vaut 2π ; sinon cette somme vaut exactement zéro (figure 4.4).

L'algorithme complet sera de la forme suivante :

```
test (A, B)
{ pour toute arête a appartenant à A et pour tout triangle b appartenant à B
  {
       si test_arete_triangle(a,b) = vrai
       {
          renvoie vrai;
       }
  }
  pour toute arête a appartenant à B et pour tout triangle b appartenant à A
  {
       si test_arete_triangle(a,b) = vrai
       {
          renvoie vrai;
       }
```

```
  }
  renvoie faux;
}
```

où la fonction "test_arete_triangle" est calculée en utilisant la méthode vue ci-dessus.

4.3 Les objets fermés

Pour les objets fermés, non nécessairement convexe, nous pouvons définir un espace intérieur et extérieur à l'objet. La collision de deux objets fermés peut être définie comme : deux objets A et B sont en collision si et seulement si il existe un point p qui appartienne à la fois à A et à B.

Le coeur de l'algorithme utilisant cette définition est le test qui vérifie si un point p est à l'intérieur d'un polyèdre P ou pas. Ce problème de test peut être résolu de deux manières : une utilise la notion d'angle solide et l'autre d'intersection de rayon (segment de droite).

La première est juste une extension de l'algorithme qui vérifie si un point est à l'intérieur d'un triangle (vu dans la section 4.2.3 page 38). En effet, en 2D il faut vérifier que la somme des angles vaut 2π, de manière similaire en 3D, il faut vérifier que la somme des angles solides vaut 4π. Les angles solides sont mesurés en stéradians et l'angle solide d'une sphère complète vaut 4π, ce qui correspond à l'aire d'une sphère unité.

L'angle solide d'un tétraèdre de sommet q et de base T est l'aire de la sphère unité S, centré en q, se trouvant à l'intérieur du tétraèdre (figure 4.5).

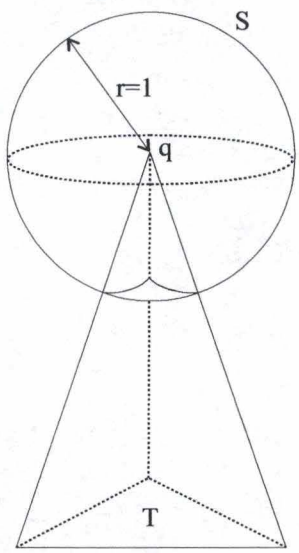


FIG. 4.5 – L'angle solide d'un tétraèdre de sommet q et de base T est égal à l'aire grisée

Si les angles solides formés par p et chacune des faces du polyèdre P sont additionnés et que le résultat de cette somme vaut 4π alors p est à l'intérieur de P, par contre, si cette somme vaut 0 alors p est à l'extérieur de P. Cet algorithme, comme son équivalent 2D, est très peu efficace car il nécessite beaucoup de calculs trigonométriques.

La deuxième méthode est nettement plus efficace. D'après des tests effectués par [O'R98], elle serait 25 fois plus rapide que la précédente. Son principe est le suivant, un point p est à l'intérieur de P si et seulement si un rayon partant de p et allant vers l'infini coupe la surface de P un nombre impair de fois (figure 4.6). Un tel rayon peut être simulé par un long segment pq, suffisamment long pour que q soit à l'extérieur de P. L'aspect problématique de cette approche est de développer une méthode pour compter avec précision le nombre d'intersection, en présence d'une grande variété de dégénérescence pouvant apparaître (figure 4.6 (b) rayon en pointillé). Par exemple, le segment pq peut toucher un sommet, être dans une face de P, etc. Nous allons générer aléatoirement un rayon et vérifier la présence éventuelle de dégénérescences. S'il n'y en a pas, alors nous pouvons compter sans crainte le nombre d'intersection entre pq et P. Si une dégénérescence est trouvée alors nous abandonnons le rayon choisi et nous en générons un autre.

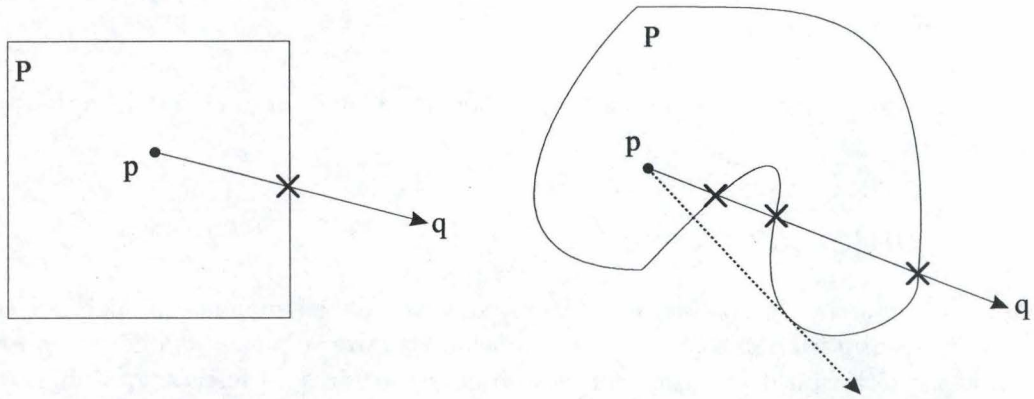


FIG. 4.6 – (a) le rayon pq intersecte P en un seul point, (b) le rayon pq intersecte P en trois points. Le rayon en pointillé est éliminé à cause de la présence d'une dégénérescence.

Regardons comment ce segment est construit. Soit d la longueur de la diagonale de la boîte limite englobant P. Lors d'un test, un segment d'une longueur d à partir de p jusque q, point situé aléatoirement sur la surface d'une sphère de rayon d et de centre p, est généré. Si ce segment intersecte un nombre impair de triangles de P alors nous pouvons affirmer que p est à l'intérieur de P.

Pour détecter une dégénérescence, nous allons améliorer notre fonction qui teste si un segment est en intersection avec un triangle, afin qu'elle nous renvoie des informations sur la position du segment. Par exemple, elle nous dira si le segment se trouve dans le plan qui contient le triangle, si le segment est en intersection uniquement avec un sommet du triangle, si le segment est colinéaire avec un côté du triangle, etc. Ses informations nous serons utiles pour détecter une dégénérescence.

L'algorithme complet de détection de collision sera de la forme suivante :

```
test (A, B)
{ pour tout sommet s appartenant à A
  {
       si test_point_dans_polyedre(s,B) = vrai
       {
          renvoie vrai;
       }
  }
  pour tout sommet s appartenant à B
  {
       si test_point_dans_polyedre(s,A) = vrai
       {
          renvoie vrai;
       }
  }
  renvoie faux;
}
```

où la fonction "test_point_dans_polyedre" est calculée en utilisant une des deux méthodes vues ci-dessus.

4.4 Les objets convexes

Nous restreignons le problème aux objets convexes ou autrement dit aux polyèdres convexes, c'est-à-dire les polytopes. Les polytopes nous offres d'autres manières d'aborder le problème de détection de collision que les objets arbitraires ou fermés. Les objets sont considérés comme l'enveloppe convexe de leurs sommets.

Dans ce contexte, nous pouvons reformuler la condition de collision comme suit : deux objets A et B ne sont pas en collision si et seulement si il existe un plan π tel que tous les sommets de A sont d'un côté et tous les sommets de B sont de l'autre côté. Un tel plan est appelé *plan séparateur*, et A et B sont dit *linéairement séparable*. De manière plus formelle,

soient a^i $(i = 1, .., n)$ et b^j $(j = 1, .., m)$ les sommets de A et B respectivement. A et B sont linéairement séparable ssi

$$\begin{aligned} &\exists w \in \mathbb{R}^3, w_0 \in \mathbb{R} : \forall i, j \quad a^i \cdot w - w_0 > 0, \quad b^i \cdot w - w_0 < 0 \\ &\Leftrightarrow (a^i, -1) \cdot (w, w_0) > 0, \quad (b^i, -1) \cdot (w, w_0) < 0 \\ &\Leftrightarrow (a^i, -1) \cdot (w, w_0) > 0, \quad (-b^i, 1) \cdot (w, w_0) > 0 \end{aligned} \tag{4.1}$$

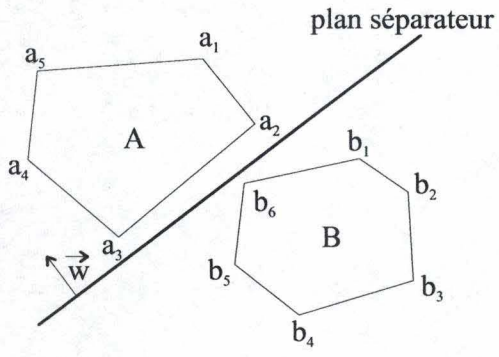


FIG. 4.7 – Deux polytopes disjoints peuvent toujours être séparés par un plan

L'algorithme que nous allons analyser, qui est un algorithme probabiliste s'inspirant des réseaux de neurones, va utiliser en entrée un ensemble

$$Z = \{(a^i, -1), (-b^j, 1), i = 1, .., n \text{ et } j = 1, .., m\} \subseteq \mathbb{R}^4$$

où les a^i et b^j sont les sommets de A et B respectivement. Il va rechercher un vecteur v grâce à la dernière définition de linéairement séparable (inégalités 4.1), ce vecteur si il existe sera égal à (w, w_o)[1]. Il s'arrêtera lorsque pour tous les z appartenant à Z, $z \cdot v > 0$, ce qui est équivalent aux inégalités 4.1.

Le principe de l'algorithme est de faire "pivoter" le plan séparateur un tout petit peu lorsqu'il trouve un point z qui est encore du mauvais côté (figure 4.8). En fait, chaque point trouvé du mauvais côté tire le vecteur normal au plan vers lui.

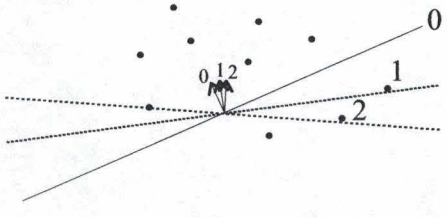


FIG. 4.8 – Chaque point trouvé du "mauvais" côté attire la normale du plan vers lui.

Au départ, v^0 est un vecteur arbitraire, disons $(0, 0, 0, 1)$ et $l = 0$ est le nombre d'itération. A la première étape, si il existe un $z \in Z$ tel que $z \cdot v^l < 0$ alors $v^{l+1} = v^l + \eta \cdot z$, où η est le poids que nous accordons au point z trouvé, et l est incrémenté de 1. Ensuite à la seconde étape, si pour tout $z \in Z$ $z \cdot v^l > 0$ alors nous avons trouvé un plan séparateur et c'est terminé, sinon il faut reprendre à l'étape une.

[1]L'équation du plan nous est donnée par l'équation $(x, y, z) \cdot w - w_0 = 0$

Le poids η est un paramètre qu'il nous faut choisir avant d'exécuter l'algorithme. Il est soit fixe ou variable, c'est-à-dire il évolue au cours des mise à jour du plan séparateur. Par exemple, si nous choisissons une valeur initial pour le poids η, nous pouvons le faire décroître au fur et à mesure des mise à jour en le mutipliant par un facteur constant (0,97 est un bon choix), ou bien, le poids η peut être calculé sur base de la "mauvaise qualité" du plan séparateur actuel par rapport au point z qui est du mauvais côté, ainsi une mise à jour possible serait $v^{l+1} = v^l + \frac{v^l z}{\|v^l\|\|z\|} \cdot z$.

L'algorithme tel qu'il est présenté dans ces grandes lignes ci-dessus a un sérieux problème de terminaison. En effet, si les deux objets sont en collision, alors l'algorithme boucle indéfiniment. Une solution est de stopper le bouclage après un certain nombre d'itérations (disons 1000), et si l'algorithme n'a pas trouvé de plan séparateur, alors nous supposons que les deux objets ne sont pas linéairement séparables. L'algorithme est à cause de cela légèrement *biaisé* vers "non linéairement séparable". Quand l'algorithme renvoie comme résultat "non linéairement séparable", il y a une petite chance que le résultat soit faux. Par contre, s'il répond "linéairement séparable", cette réponse est toujours correcte.

L'algorithme que nous venons de voir à un gros avantage, c'est qu'une situation de non-collision est très rapidement déterminée. Par contre, il a un gros désavantage, c'est qu'il ne fournit pas de "témoin" de la collision (i.e., une arête ou un polygone).

Il existe deux autres manières de reformuler la condition de collision, une qui est fort dans la même idée que la précédente, utilisant le concept d'*axe séparateur*. Cela permet de construire un autre algorithme tout aussi efficace et non biaisé. Cette condition est : deux objets A et B ne sont pas en collision si et seulement si il existe un axe séparateur c'est-à-dire un axe qui lorsque l'on projète les objets A et B sur lui donnent des intervalles qui ne se chevauchent pas. Nous aurons l'occasion d'analyser cet algorithme dans la section 4.5.4 page 53 de ce chapitre.

L'autre est basée sur la notion de distance : deux objets convexes A et B ne sont pas en collision si et seulement si la distance Euclidienne minimum les séparant est plus grande que zéro. La distance entre ces deux objets est donc la plus petite distance Euclidienne d_{AB} :

$$d_{AB} = \inf_{p \in A,\, q \in B} \|p - q\|$$

L'idée à la base de cette approche est de maintenir la paire de composants[2] les plus proches entre une paire d'objets, notés respectivement A et B. Puisque, en général, les objets se déplacent lentement entre deux étapes de temps, ces composants seront sans doute encore les plus proches à l'étape suivante (image suivante de l'animation) ou, si ce n'est pas le cas, nous les trouverons dans le voisinage des anciens.

Les polyèdres doivent passer par une phase de précalcul afin que nous puissions être capables de maintenir une paire de composants les plus proches de manière efficace. À cette fin, la région Voronoï de chaque composant est construite. La région de Voronoï

[2]un composant est soit un sommet, une arête ou un polygone

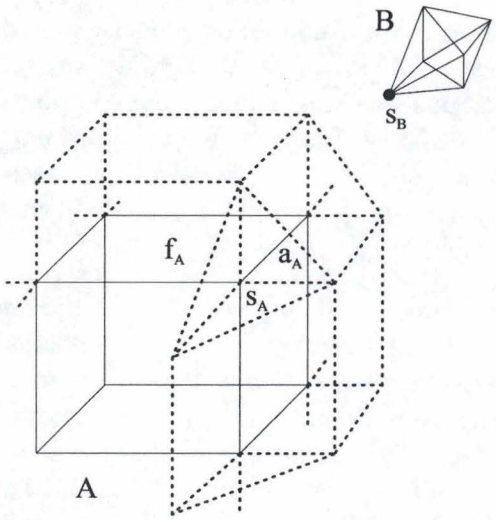


FIG. 4.9 – Un exemple d'une région de Voronoï d'un polyèdre convexe. s_B n'est pas dans la région de Voronoï de f_A, donc (s_B, f_A) ne sont pas des composants *réalisants*

d'un composant est l'ensemble des points qui lui sont les plus proches et l'ensemble de ces régions forme le diagramme de Voronoï (voir la section 1.2.6 page 10 pour une définition plus complète). Ainsi comme nous pouvons le voir sur la figure 4.9, les composants s_A, a_A et f_A ont chacun une région de Voronoï particulière à leur géométrie.

Soient (u, v) une paire de composants appartenant respectivement à A et à B, et (p_u, p_v) deux points tels que $p_u \in u$ et $p_v \in v$. Nous dirons que (u, v) et donc par là (p_u, p_v) *réalisent* la distance entre A et B si et seulement si p_u est dans la région de Voronoï de v et si p_v est dans la région de Voronoï de u.

Si nous avons deux composants (f_A, s_B), nous pouvons facilement vérifier s'ils réalisent ou non la distance entre A et B (figure 4.9). Si ce n'est pas le cas, nous pouvons retrouver un autre composant (soit appartenant à A ou B, quel que soit celui qui a fait échouer le test) qui est le plus proche de l'autre. Ce composant le plus proche est incident de celui qui a été rejeté, par exemple pour notre exemple, prenons a_A qui est une arête frontière du polygone f_A. Nous affinons notre recherche ainsi jusqu'à trouver un composant qui est le plus proche. Puisque l'objet en question est convexe, nous obtiendrons un optimum global (pas simplement un local).

Pour plus d'informations sur cet algorithme, notamment au sujet du calcul de la région de Voronoï d'un polyèdre, des dégénérescences possibles, etc., nous pouvons nous référer à la thèse [Lin93] qui est en très grande partie consacrée à ce sujet.

4.5 Méthode hiérarchique

Les volumes limites sont utilisés dans presque tous les domaines de l'informatique graphique pour accélérer les calculs, si possible en ne les faisant pas si le résultat peut être aisément obtenu par une pré-vérification. Dans ce contexte, nous voudrions avoir seulement à faire avec les volumes limites, parce que traiter directement avec les objets eux-mêmes est beaucoup trop coûteux. Pour gagner de la vitesse, le volume limite doit être beaucoup plus simple que l'objet qu'il englobe. En même temps, cela constitue un gros désavantage puisque suivant la géométrie de l'objet englobé, le volume limite peut contenir énormément d'espace vide.

En résumé, voici les quelques caractéristiques les plus souhaitées des volumes limites :
- facile à calculer
- besoin en mémoire peu élevé
- rapidement transformable
- test de chevauchement simple
- ajuste l'objet englobé au maximum

Nous supposons comme entrée un ensemble S de n primitives géométriques[3] qui définissent la frontière d'un objet polygonal arbitraire. Un arbre à volumes limites (AVL) est un arbre, $AVL(S)$, qui spécifie une hiérarchie de volumes limites sur S. A chaque noeud, v, de $AVL(S)$ correspond un sous ensemble, $S_v \subseteq S$, et le noeud racine est associé à tout l'ensemble S. Chaque noeud interne ($\neq$ une feuille) a deux ou plusieurs fils et le nombre maximum de fils pour chaque noeud interne est appelé le *degré* de l'arbre, noté δ. L'union de tous les sous-ensembles associés aux fils de v est égal à S_v. A chaque noeud est associé un volume limite, $b(S_v)$, qui est une approximation de l'espace occupé par S_v en utilisant une classe de volumes spécifiée (exemple : boîte, sphères, etc.)

Dans cette section, nous allons nous focaliser sur le cas d'un objet simple (rigide), défini par un ensemble V de primitives géométriques (triangles), étant donné une certaine position et orientation, qui se déplace dans un environnement, défini par un ensemble E de primitives géométriques d'obstacles (triangles). Nous avons donc une hiérarchie de l'objet en mouvement et une hiérarchie de l'environnement constituant notre paire d'objets.

4.5.1 Critères de conception

Le problème le plus important des algorithmes de détection de collision basés sur un arbre à volumes limites est la conception d'une structure de données optimale. Actuellement, il existe quatre classes de volumes limites étant les plus courantes, comprenant les sphères, les boîte limites à axes alignés (A*xis* A*ligned* B*ounded* B*ox*), les boîtes limites orientées (O*riented* B*ounded* B*ox*) et les polytopes à orientation discrète (k-D*iscrete* O*riented* P*olytope*). Le choix d'une classe plutôt qu'une autre, à utiliser comme volume limite dans un arbre, dépend du domaine de l'application et des différentes contraintes

[3] généralement des polygones triangles en trois dimensions

qui lui sont inhérentes. Par exemple, en lancer de rayon (ray tracing), les volumes limites choisis doivent englober le plus juste possible les objets, mais aussi permettre des tests d'intersection très efficaces entre les rayons et les volumes limites.

Malheureusement, la conception d'une structure de donnée optimale basée sur une analyse théorique est extrêmement difficile, parce que nous sommes obligés de faire des hypothèses statistiques fortes sur la distribution de la géométrie et la position des modèles en entrée. C'est pourquoi, les chercheurs ont appliqué quelques critères pratiques pour évaluer les volumes limites existants.

Etant donné deux larges objets en entrée et leur hiérarchie de volumes limites construite pour les approximer, le coût total[4] pour vérifier si deux objets sont en collision est quantifié par

$$T = N_v \times C_v + N_p \times C_p$$

où T est la fonction de coût total pour une détection de collision, N_v est le nombre de pairs de volumes limites testés pour chevauchement (intersection), C_v est le coût du test de chevauchement d'une paire de volumes limites, N_p est le nombre de primitives géométriques ayant subi un test d'intersection (des triangles dans la plupart des applications) et C_p est le coût du test de contact entre paires de primitives.

Bien que l'équation ci-dessus soit une mesure raisonnable du coût associé à l'exécution d'un test de détection de collision, elle ne prend pas en compte le coût de *mise à jour* de la hiérarchie, maintenant un volume limite approprié pour les tests de chevauchement, lorsque l'objet est en rotation ou translation. Donc, nous proposons que le coût total du processus de détection de collision en environnement dynamique soit calculé par

$$T = N_v \times C_v + N_p \times C_p + N_u \times C_u$$

où T, N_v, C_v, N_p et C_p sont définis comme plus haut, N_u est le nombre total de noeuds devant être mis à jour et C_u est le coût de mise à jour de tel noeud.

Deux autres facteurs non inclus dans cette dernière équation peuvent également influencer le choix d'une classe de volumes limites. Le premier est le temps de précalcul pour la construction de l'arbre initial. Le second est la mémoire totale requise par la structure choisie. Suivant les applications et les objets en entrée, ces deux facteurs peuvent devenir des critères décisifs pour la sélection d'une classe de volumes limites.

Etant donné la fonction de coût ci-dessus, une structure de volumes limites doit :

1. approximer les primitives d'entrées le plus précisément possible (pour diminuer N_v, N_p et N_u) ;
2. être soutenue par un test de chevauchement rapide entre deux volumes limites (pour diminuer C_v) ;
3. être mise à jour aussi vite que possible (pour diminuer C_u) ;

[4] généralement exprimé en ms

4. être construite efficacement avec un minimum de mémoire.

Malheureusement, ces exigences sont habituellement en conflit. Par exemple, utiliser des enveloppes convexes comme volumes limites nous permettent d'avoir des N_v, N_p et N_u minimaux, mais les coûts élevés de C_v et C_u rendent ce type de volumes limites un mauvais choix. A l'autre extrême, pour les sphères C_u vaut zéro et C_v est également très bas. Cependant, cela requiert un N_v très grand. Il est clair qu'aucune représentation hiérarchique ne donne les meilleurs performances tout le temps.

Le défi majeur est de trouver le meilleur compromis entre chacune de ces exigences.

4.5.2 Approche Top-Down contre Bottom-Up

Lorsque nous construisons un AVL d'un ensemble, S, de primitives géométriques, nous pouvons le faire de deux manières, une appelée top-down ou *descendante*, l'autre bottom-up ou *ascendante*.

- Une approche *ascendante* commence avec les primitives géométriques comme les feuilles de l'arbre et tente de les regrouper ensemble récursivement (en utilisant des informations locales), jusqu'à atteindre un unique noeud principal qui approxime l'ensemble S tout entier.
- Une approche *descendante* débute avec un noeud qui approxime S, et utilise des informations basées sur l'ensemble entier pour diviser récursivement le noeud jusqu'à atteindre les feuilles de l'arbre.

4.5.3 Degré de l'arbre

Minimiser la hauteur de l'arbre est une qualité souhaitable quand on construit une hiérarchie, ainsi lorsque des recherches sont effectuées, nous pouvons traverser l'arbre, de la racine à une feuille, en un petit nombre d'étapes. Le degré, δ, spécifie le nombre maximum de fils qu'un noeud peu avoir. Typiquement, plus le degré est élevé, plus la hauteur de l'arbre est faible. Il est ici aussi question d'un compromis entre des arbres de haut et faible degré.

Un arbre avec un degré élevé tend à être petit en hauteur, mais beaucoup plus de temps sera consacré par noeud lors de la recherche. D'un autre côté, une arbre de faible degré a une grande hauteur et moins de temps est consacré par noeud lors de la recherche.

Les arbres binaires ($\delta = 2$) sont les plus avantageux et ceci pour deux raisons :

1. il sont simple et rapide à calculer, puisque il y a moins d'opérations à effectuer lorsqu'on découpe un ensemble en deux morceaux que lorsqu'on découpe un ensemble en trois ou plusieurs morceaux.
2. des évidences analytiques nous suggèrent que les arbres binaires sont meilleurs que les arbres avec un $\delta > 2$. En particulier, si nous considérons un arbre balancé (avec n feuilles) dont le degré de ses noeuds interne est plus grand que deux, alors la quantité de travail nécessaire pour une recherche partant de la racine jusqu'à une feuille est

proportionnel à $f(\delta) = (\delta - 1)\log_\delta n$, puisque au plus $\delta - 1$ des δ fils à besoin d'être testé avant de savoir par où descendre. De simples calculs montrent que la fonction $f(\delta)$ est monotone croissante sur l'intervalle $\delta \in (1, \infty)$ (et $f'(1) = 0$). Donc, si nous nous restreignons aux valeurs entières de δ plus grandes que un, nous constatons que $f(\delta)$ est minimisée par $\delta = 2$. Bien évidemment, cette analyse ne prend pas en compte que généralement les recherches effectuées dans un AVL ne consiste pas toutes en un simple chemin racine-feuille. Mais au vu des AVL existant, le choix de $\delta = 2$ est raisonnable et justifié.

Nous allons voir dans ce qui suit quelques uns des volumes limites les plus couramment utilisés.

4.5.4 AABB - Boîte limite à axes alignés

Nous allons voir un système de détection de collision qui se base sur une représentation hiérarchique d'un objet en utilisant des boîtes limites à axes alignés. Dans les arbres AABB, les boîtes sont alignées sur les axes du système de coordonnées local de l'objet, et donc toutes les boîtes d'un arbre ont la même orientation.

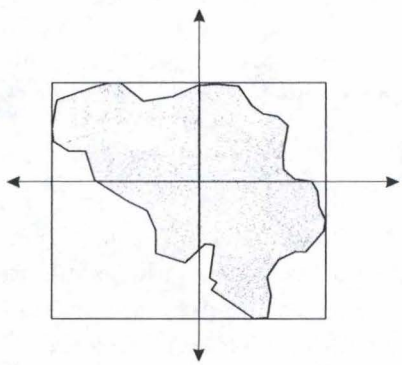


FIG. 4.10 – Une vue bidimensionnelle de la AABB d'un objet

Construction d'une AABB

Une AABB requiert pour son stockage en mémoire 6 scalaires, 3 pour la position du centre et 3 pour la dimension. Soit S un ensemble de n primitives géométriques (triangles). Chaque primitive est constitué de trois sommets p_i, q_i et $r_i \in \mathbb{R}^3$. Pour construire une AABB d'un tel ensemble, il faut calculer à la fois le point minimum et maximum de cet ensemble de $3n$ points, notons-les m et M respectivement. Ceci ce fait très simplement en calculant le minimum (resp. le maximum) pour chacune des coordonnées, notons-les (x_j, y_j, z_j), de tous les points de l'ensemble.

Nous calculons m et M,

$$m = (\min_{j\in\{1,..,3n\}}(x_j), \min_{j\in\{1,..,3n\}}(y_j), \min_{j\in\{1,..,3n\}}(z_j))$$

$$M = (\max_{j\in\{1,..,3n\}}(x_j), \max_{j\in\{1,..,3n\}}(y_j), \max_{j\in\{1,..,3n\}}(z_j))$$

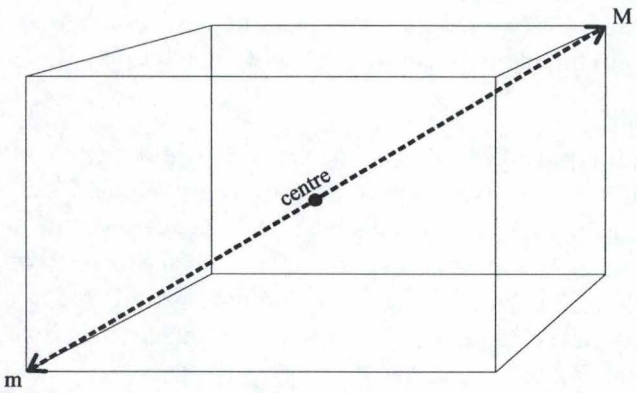


FIG. 4.11 – Représentation d'une AABB

Une fois ces deux points connus, nous pouvons facilement calculer le centre et la dimension de la boîte,

$$centre = \frac{m + M}{2}$$

$$dimension = \frac{M - m}{2}$$

Construction de l'arbre

L'arbre AABB que nous allons construire est en fait un arbre binaire. Nous allons construire cet arbre de manière descendante (top-down), par des subdivisions récursives. A chaque pas de récursion, une AABB plus petite est calculée sur l'ensemble de primitives et cet ensemble est ensuite divisé en deux, en choisissant un plan de partitionnement bien choisi. Ce processus se poursuit jusqu'à ce que chaque sous-ensemble ne contienne plus qu'un élément. Donc, un arbre AABB d'un ensemble de n primitives est constitué de n feuilles et $n - 1$ noeuds internes.

Comment va-t-on choisir ce plan de partitionnement ? A chaque étape, nous choisissons un plan de partitionnement orthogonal au plus long axe de la AABB. De cette manière, nous obtenons une subdivision dite "large". En général, les AABB larges, c'est-à-dire qui ressemble à des cubes plutôt qu'à des fins rectangles, donne de meilleure performance pour les tests de chevauchements, en effet, une boîte donnée chevauchera moins de grosses boîtes (figure 4.12 (a)) que de fines boîtes (figure 4.12 (b)).

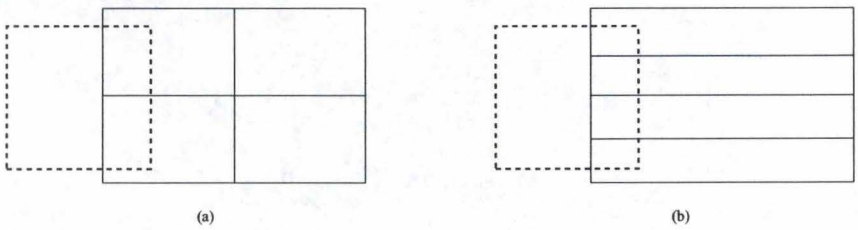


FIG. 4.12 – Après 2 pas de récursion : (a) partitionnement orthogonal au plus long axe de la AABB, (b) partitionnement orthogonal à un axe quelconque

Nous allons positionner de plan de partitionnement le long du plus long axe, en choisissant ϕ, la coordonnée sur cet axe du plan de partitionnement. L'ensemble des primitives géométriques sera donc divisé en un sous-ensemble négatif et un sous-ensemble positif correspondant respectivement au sous-espace du plan. Une primitive sera classée comme positive si le milieu de sa projection sur l'axe est plus grand que ϕ et négatif autrement (figure 4.13). Par cette méthode, le degré de chevauchement entre les AABB de deux sous-ensembles est maintenu faible.

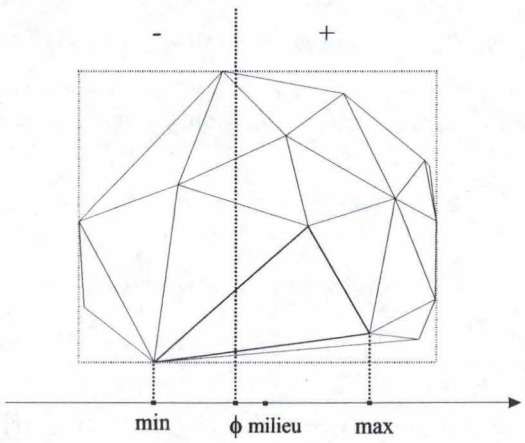


FIG. 4.13 – Une primitive est classée comme positive si le milieu de sa projection sur l'axe est plus grand que ϕ

Pour le choix de ϕ, plusieurs heuristiques sont possibles. La première est de choisir tout simplement la médiane de la AABB pour ϕ, nous divisions donc la boîte en deux moitiés égales. Une autre est de fixer ϕ de tel sorte que les deux sous-espaces engendrés soient de même taille. Nous obtenons ainsi un arbre équilibré optimal. Il est possible d'imaginer d'autres méthodes pour choisir ϕ, le but étant de minimiser le chevauchement des sous-ensembles des AABB projetés sur l'axe le plus long. En fait, si nous reprenons les notations du point 4.5, il y a exactement $\frac{1}{2}(2^{|S_v|} - 2)$ manières de faire pour partitionner en deux un ensemble S_v de primitives géométriques à l'aide d'un plan.

Occasionnellement, il peut arriver que toutes les primitives se trouvent classées dans un même côté du plan. Cela se produit lorsque l'ensemble de primitives est très petit. Dans ce cas, une division en deux ensembles de même taille résout le problème, nous ne faisons alors plus attention à la position des primitives géométriques.

Test d'intersection

Le test d'intersection (ou de chevauchement) entre deux objets est réalisé en testant chaque paire de noeuds récursivement. Pour chaque paire de noeuds visitée, les AABB associés sont testées pour voir s'ils se chevauchent. Uniquement les noeuds pour lesquels les AABB se chevauchent sont parcourus plus en avant. Si les deux noeuds en question sont des feuilles, alors les primitives sont testées pour intersection, et le résultat du test est transmis. Sinon, si un des noeuds est une feuille et l'autre un noeud interne, alors la primitive associée à cette feuille sera testée pour intersection avec chaque fils du noeud interne. Finalement, si les deux noeuds sont des noeuds internes, alors le noeud avec le plus petit volume est testé pour intersection avec les fils du noeud de volume plus large.

Puisque les systèmes de coordonnées locaux d'une paire d'objets peuvent être arbitrairement orienté, nous avons besoin d'un test de chevauchement pour des boîtes orientées. Un algorithme très rapide pour tester le chevauchement de deux boîtes orientées est connu sous le nom de *test des axes séparateurs* ou SAT (separating axes test). Un axe séparateur de deux boîtes est un axe pour lequel les projections des boîtes sur celui-ci ne se chevauchent pas. L'existence d'un tel axe pour une paire de boîtes est suffisante pour classer les boîtes comme disjointes. Il est possible de montrer [Got96] que pour toute paire de polytopes tridimensionnels disjoints, il existe un axe séparateur qui est soit orthogonal à une face d'un des polytopes, ou soit orthogonal à une paire d'arêtes prisent sur chaque polytope.

Cela constitue en tout 15 axes séparateurs potentiels qu'il faut tester pour une paire de boîtes orientées (3 orientations de face par boîte plus 9 combinaisons de direction d'arête). En effet, notons par A_i $(i = 1, .., 3)$ et B_j $(j = 1, .., 3)$ les directions des axes de coordonnées de chacune des AABB respectivement. Les axes séparateurs qu'il faut tester sont dans les directions suivantes :

$$
\begin{array}{ll}
A_1 & A_1 \times B_1 \\
A_2 & A_1 \times B_2 \\
A_3 & A_1 \times B_3 \\
B_1 & A_2 \times B_1 \\
B_2 & A_2 \times B_2 \\
B_3 & A_2 \times B_3 \\
 & A_3 \times B_1 \\
 & A_3 \times B_2 \\
 & A_3 \times B_3
\end{array}
$$

Si aucuns des axes ne séparent les boîtes, alors les boîtes se chevauchent. Il est important de constater que l'algorithme a besoin de calculer l'orientation relative, représentée par une matrice 3x3, d'une boîte par rapport à l'autre et ensuite sa valeur absolue avant d'effectuer

le test sur les 15 axes. Il ne faut pas oublié que chaque boîte est exprimée dans le système de coordonnées local de l'objet auquelle elle est associée. Comme dans les arbres AABB l'orientation relative est la même pour chacune des paires de boîtes testées, cette matrice ne devra être calculée qu'une seule fois.

Les objets déformables

Les arbres AABB se prêtent tout à fait bien à être utilisé pour des objets déformables. Dans ce contexte, un objet déformable est un ensemble de primitives dans lequel les positionnements et les formes des primitives, dans le système de coordonnée local à l'objet, change au cours du temps. Un exemple typique d'un objet déformable est un polyèdre dans lequel les coordonnées locales des sommets dépendent du temps.

Au lieu de reconstruire l'arbre tout entier après chaque déformation, il est beaucoup plus rapide de redimensionner les boîtes de l'arbre. Nous allons réaliser ce redimensionnement de manière ascendante (bottom-up). Soient S l'ensemble des primitives et S^+, S^- des sous-ensembles de S tel que $S^+ \cup S^- = S$, et soient B^+, B^- les plus petites AABB de S^+ et S^- respectivement et B, la plus petite boîte englobant $B^+ \cup B^-$. Par ce fait, B est aussi la plus petite AABB englobant S. Cette propriété va nous être très utile pour notre méthode de redimensionnement.

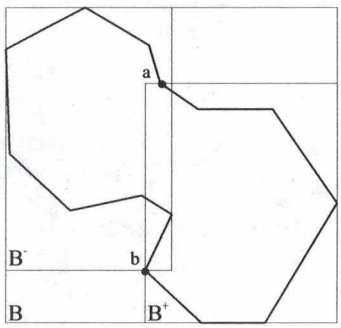
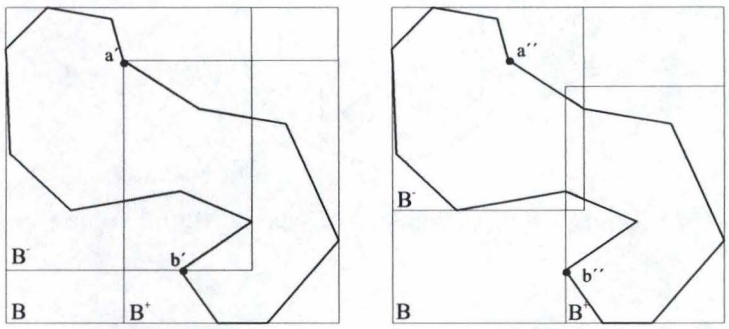


FIG. 4.14 – La plus petite AABB d'un ensemble de primitives englobe les plus petites AABB des sous-ensembles des partitions de l'ensemble (2D)

Comment allons nous faire ce redimensionnement ? D'abord, les boîtes limites des feuilles sont recalculées, après cela, chaque parent est recalculé en utilisant les boîtes de leurs fils dans un ordre ascendant strict. Donc pour chaque noeud interne, les fils sont d'abord visités, puis la boîte limite est recalculée. Grâce à cette méthode, un arbre AABB est redimensionné en un temps linéaire par rapport au nombre de noeuds. Des expérimentations effectuées par [Ber98] montre que pour des objets composé de plus de 6000 triangles, le redimensionnement d'un arbre AABB est environ dix fois plus rapide que de le reconstruire.

Tout cela à l'air bien joli, mais derrière cette méthode ce cache un inconvénient. A cause des changements de position des primitives de l'objet après une déformation, les

boîtes d'un arbre redimensionné ont un plus grand degré de chevauchement que celles d'un arbre reconstruit (figure 4.15). Un plus grand degré de chevauchement implique que plus de noeuds devront être visités lors d'un test d'intersection, les performances du test s'en trouveront réduites. Heureusement, cet inconvénient est très peut visible sauf lorsque l'objet subit des torsions excessives, des redimensionnements aléatoires, etc.



(a) Redimensionné (b) reconstruit

FIG. 4.15 – Redimensionner contre reconstruire l'arbre après une déformation

4.5.5 Sphère

Les sphères sont les secondes en popularité. La géométrie de la sphère est des plus simples, ce qui la rend très attrayante. La transformation des sphères est très simple et nous n'avons plus le problème de l'alignement aux axes.

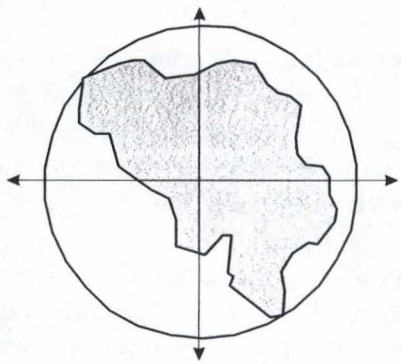


FIG. 4.16 – Une vue bidimensionnelle de la sphère englobante d'un objet

Construction d'une sphère

Une sphère requiert pour son stockage en mémoire 4 scalaires, 3 scalaires pour la position du centre et un scalaire pour le rayon. Pour construire une sphère qui englobe le mieux un

objet, plusieurs approches sont possibles suivant la précision souhaitée.

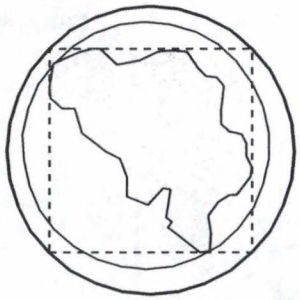


FIG. 4.17 – Différents niveaux de précision pour le calcul d'une sphère englobante (coupe 2D)

Soit S un ensemble de n primitives géométriques (triangles). Chaque primitive est constitué de trois sommets p_i, q_i et $r_i \in \mathbb{R}^3$. Ce qui nous fait en tout $3n$ points.

Une méthode consiste à calculer m et M comme pour les AABB, et ensuite à calculer

$$centre = \frac{m + M}{2}$$

$$rayon = ||M - centre||$$

Nous obtenons ainsi la sphère en gras sur la figure 4.17.

Construction de l'arbre

Une simple sphère ne peut approximer précisément la forme de la plupart des objets. Par contre, une union de sphères se chevauchant partiellement peut donner une meilleure approximation (figure 4.18). En règle générale, une union impliquant beaucoup de sphères est capable approximer le contour d'un objet avec plus de précision qu'une union de quelques sphères. Le nombre de sphère dans une union détermine donc une balance entre vitesse et exactitude.

L'arbre de sphère que nous allons construire n'est plus un arbre binaire, comme pour les arbres AABB, mais un arbre quelconque. Nous allons construire cet arbre de manière descendante (top-down), par des subdivisions récursives.

La racine de l'arbre (de la hiérarchie) est la sphère qui englobe l'objet tout entier. Le premier niveau de hiérarchie est une union qui contient le plus petit nombre de sphères. Dans les niveaux suivant le nombre de sphères des unions va s'accroître. Il n'y a aucune raison que les parents aient nécessairement le même nombre de fils, cependant une valeur limite permet de simplifier la structure de donnée des noeuds de l'arbre.

Une chose très importante à souligner est que les fils de chaque parent couvrent toutes les parties de l'objet que le parent couvre lui-même. Aussi longtemps que les fils d'une sphère

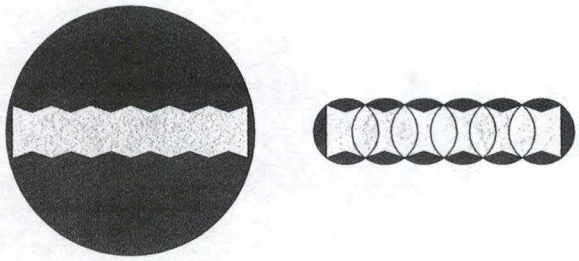


FIG. 4.18 – L'union de six cercles à droite donne une meilleur approximation qu'un simple cercle à gauche

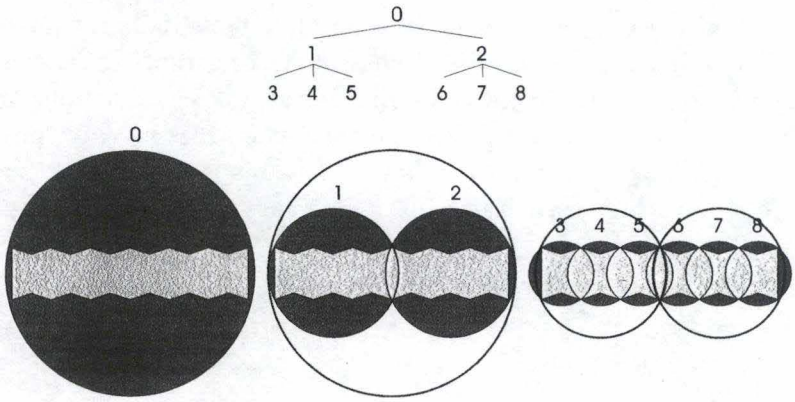


FIG. 4.19 – Une hiérarchie de trois niveau. Pour chaque niveau, le niveau précédent est laissé en pointillé

couvrent toutes les parties de l'objet qu'elle couvre, les fils sont autorisés à couvrir plus que ce qu'ils ne doivent (figure 4.19 sphère 5). Cette zone de recouvrement est bien évidemment redondante puisqu'elle est déjà couverte par un des parents frères et donc par ses fils. La hiérarchie sera d'autant plus efficace si les fils couvrent peu l'extérieur de leur parent. En revanche, cette situation de couverture supplémentaire est assez intéressante puisque sans elle, des parties proches des frontières d'un parent ne pourrait pas être couverte. Donc, cette situation donne une meilleur approximation.

Nous allons voir une méthode qui permet de construire une hiérarchie de sphère englobant au mieux un objet. Cette méthode est basée sur les arbres octaux. Un arbre octal est une structure de donnée qui représente une subdivision récursive de l'espace tridimensionnelle. Cette subdivision de l'espace va nous être utile pour calculer les positions des sphères.

Regardons comment fonctionne un algorithme construisant un arbre octal. Tout d'abord, la racine de l'arbre (niveau 0) est définie comme la plus petite boîte qui englobe tout l'objet.

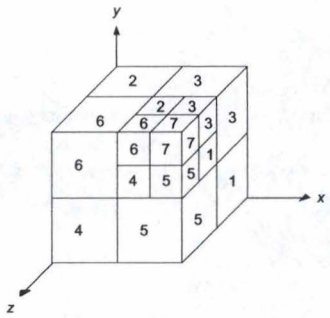


FIG. 4.20 – Découpe octal d'une boîte

Ensuite, pour construire les fils de la racine (niveau 1), la boîte de la racine est découpée en huit morceaux plus petits, appelés *octants*. A chacun de ses octants est associé un ensemble de primitives géométriques y étant contenues. Si cet ensemble est vide pour un octant, alors celui-ci disparaît et la racine a un fils de moins. Finalement, les niveaux suivant de l'arbre sont construits en appliquant récursivement sur chaque fils la méthode ci-dessus. La partie la plus difficile de l'algorithme est l'étape qui détermine l'appartenance de polygones à un fils.

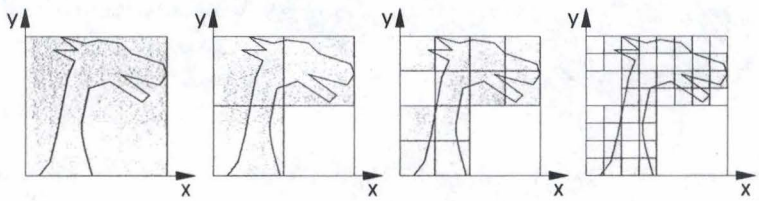


FIG. 4.21 – Les quatre premiers niveaux d'un arbre octal d'un objet 2D. Les noeuds sont grisés.

Maintenant que nous avons un arbre octal d'un objet, une méthode simple pour construire l'arbre de sphère est de mettre une sphère autour de chaque noeud de l'arbre octal. L'arbre de sphère maintient les mêmes relations parent-enfant que l'arbre octal. Un ensemble de fils couvre avec certitude tout ce que leur parent couvre parce que les sphères circonscrivent les boîtes limites.

Il peut arriver que des régions internes à l'objet ne soient pas couvertes par des sphères (figure 4.22 niveau 4). L'arbre octal n'a pas besoin de ces régions parce que ces noeuds doivent contenir une partie de la surface de l'objet, pas son volume. La caractéristique de cohérence temporelle empêche d'autres objets de " sauter " à travers la surface de l'objet pour atteindre la région non couverte.

Dans un arbre de sphère basé sur un arbre octal, les sphères de niveau j+1 ont un rayon deux fois plus petit et un volume huit fois plus petit que les sphères de rayon j.

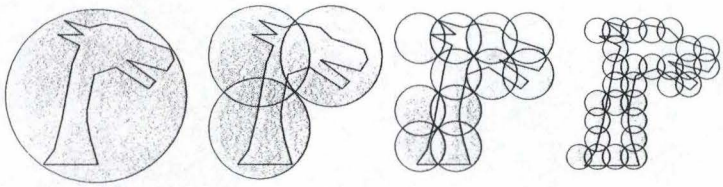


FIG. 4.22 – L'arbre octal de la figure 4.21 défini l'arbre de sphères dont les quatre premiers niveaux sont donnés.

Test d'intersection

Une paire de sphères est en intersection si la distance entre leur centre est plus petite que la somme de leur rayon (figure 4.23). Pour une sphère centrée en c_1 de rayon r_1 et une autre sphère centrée en c_2 de rayon r_2, nous devons tester si $|c_1 - c_2| < r_1 + r_2$. Cependant, $|c_1 - c_2| = \sqrt{(c_1 - c_2)(c_1 - c_2)}$, donc une approche plus efficace serait de tester si $(c_1 - c_2)(c_1 - c_2) < (r_1 + r_2)^2$.

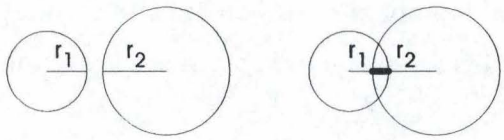


FIG. 4.23 – Deux sphères sont en intersection si la distance entre leur centre est plus petite que la somme de leur rayon (coupe 2D)

Le test d'intersection entre deux arbres de sphère, approximant chacun un objet, se fait en testant récursivement chaque paire de noeud (comme pour les arbres AABB) jusqu'à arriver au niveau des feuilles qui contiennent les primitives géométriques. Uniquement les noeuds pour lesquels les sphères associées sont en intersection sont parcourus plus en avant.

4.5.6 OBB - boîte limite orientée

Nous allons analyser un algorithme de détection de collision qui exploite la représentation hiérarchique d'un objet en boîtes limites orientées.

Construction d'une OBB

Une OBB requiert pour son stockage en mémoire 15 scalaires, 9 pour la matrice 3x3 représentant l'orientation, 3 pour la position du centre et 3 pour la dimension.

Soit S, un ensemble de n primitives géométriques (triangles). Chaque primitive est constituée de 3 sommets p^i, q^i et r^i $\forall i \in \{1, .., n\}$.

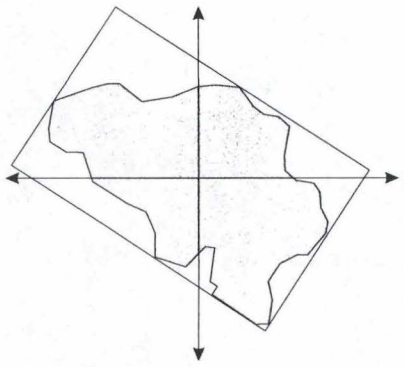


FIG. 4.24 – Une vue bidimensionnelle de la OBB d'un objet

Nous allons examiner la distribution spatiale des primitives et ensuite déterminer les deux directions dans lesquelles elles ont la plus grande et plus petite variance. Ces deux directions sont toujours perpendiculaires, en effet, c'est une des propriétés des mesures de la variance et covariance. Finalement, ces deux directions, plus une troisième perpendiculaire à elles deux, donneront l'orientation des axes de la boîte limite.

Calculons tout d'abord la moyenne μ et la matrice de covariance[5] C, dans une notation vectorielle,

$$\mu = \frac{1}{3n} \sum_{i=0}^{n} (p^i + q^i + r^i)$$

$$C = \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix}$$

$$\text{avec} \quad C_{jk} = \frac{1}{3n} \sum_{i=0}^{n} (\bar{p}_j^i \bar{p}_k^i + \bar{q}_j^i \bar{q}_k^i + \bar{r}_j^i \bar{r}_k^i), \qquad 1 \leq j, k \leq 3$$

où $\bar{p}^i = p^i - \mu$, $\bar{q}^i = q^i - \mu$ et $\bar{r}^i = r^i - \mu$. Chacun d'eux est un vecteur 3×1, c'est-à-dire $\bar{p}^i = (\bar{p}_1^i, \bar{p}_2^i, \bar{p}_3{}^i)^T$ et C_{jk} sont les éléments de la matrice de covariance 3×3 (qui est symétrique).

Les vecteurs propres d'une matrice symétrique, tel que C, sont mutuellement orthogonaux. Après les avoir normalisés, nous les utiliserons comme base. En ce qui concerne la dimension de la boîte, nous cherchons les sommets extrémaux suivant chaque axe de cette nouvelle base, et ainsi nous pouvons dimensionner la boîte limite.

[5]Etant donné un ensemble de n variables notées x_1, ..., x_n, la matrice de covariance est définie par

$$V_{ij} = cov(x_i, x_j) \equiv E\left((x_i - \mu_i)(x_j - \mu_j)\right))$$

où μ est la moyenne.

L'approche que nous venons de voir est erronée. En effet, les sommets à l'intérieur de l'objet, qui rappelons-le peut être une soupe de polygones, n'influence pas en soi le choix de la boîte limite mais par contre, ils ont une influence aléatoire sur les vecteurs propres. Par exemple, une zone très dense de sommets à l'intérieur d'un objet peut influencer l'alignement de la boîte limite en sa faveur.

Pour remédier à ce problème, nous allons travailler sur l'enveloppe convexe de l'ensemble S de primitives qui est constituée de m triangles.

Construction de l'arbre

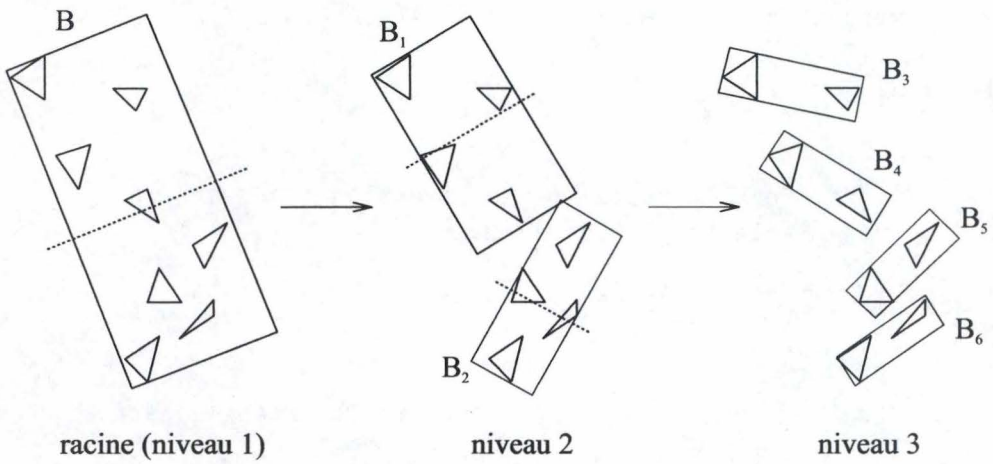


FIG. 4.25 – Les trois premiers niveaux d'une hiérarchie de OBB

Nous allons construire la hiérarchie de manière descendante sur l'ensemble de primitive S. Pour commencer construisons la racine (niveau 1) qui est une boîte limite orientée, B, englobant tout l'ensemble de primitives S (figure 4.25). Ensuite, nous découpons l'ensemble S en deux sous-ensembles disjoints S_1 et S_2. Puis nous construisons le niveau 2 de la hiérarchie à partir de ces deux sous-ensembles, ce qui nous donne les boîtes limites B_1 et B_2 qui englobent respectivement le sous-ensemble S_1 et le sous-ensemble S_2. Ces deux boîtes sont les fils de B. La procédure est appliquée récursivement, construisant ainsi l'arbre de boîtes limites. La récursion s'arrête lorsque l'on atteint le niveau des primitives, les boîtes englobant ces primitives deviennent les feuilles de la hiérarchie de boîtes limites orientées.

La complexité de cette algorithme de construction est de l'ordre de O(n log n).

Comment se déroule la découpe d'un ensemble de primitives en deux sous-ensembles ? La technique est de le découper en utilisant un plan séparateur orthogonal à un des axes de la boîte. Nous choisissons cet axe comme celui ayant son vecteur propre dont la valeur propre associée est la plus grande. C'est la direction dans laquelle l'ensemble des primitives géométriques est le plus étalé. Ici aussi, comme pour les arbres AABB, nous choisissons

une coordonnée sur l'axe par lequel le plan séparateur passe. Cette coordonnée est choisie comme étant la moyenne, μ, des sommets le long de l'axe choisi. Les primitives sont assignées du côté du plan où leur centroïde se trouve.

Regardons ce que contient comme informations un noeud de cette hiérarchie de boîtes limites orientées.

```
class boite
{
  // positionnement de la boite relativement a sa boite parente
  // boite vers espace parent: x_p = pR*x_b + pT
  // parent vers espace boite: x_b = pR.T()*(x_p - pT)
  double pR[3][3];
  double pT[3];

  // dimensions
  double d[3]; // c'est la moitie de la dimension transversale
               // +- le "rayon" de la boite

  boite *fils1;
  boite *fils2;

  tri *trp; //si une feuille alors pointe vers le triangle associe

  int feuille() { return (!fils1 && !fils2); }
  double taille() { return d[0]; }
  ...
  ...
};
```

Les variables pR et pT désignent le positionnement de la boîte par rapport à la boite parente. Le petit "p" est là juste pour rappeler que la rotation R et la translation T sont relative au système de coordonnées du parent. Ceci nous permet de maintenir le positionnement relatif des boîtes entre deux arbres lorsque nous les traversons pour tester le chevauchement. Les axes de la boîte correspondent aux vecteurs colonnes de la matrice pR.

Test d'intersection

Le test de chevauchement de deux objets A et B, représentés par leur arbre de boîtes limites, est effectué récursivement en testant chaque paire de noeud pour intersection (figure 4.26). Plutôt que de répéter ce qui a déjà été dit pour les arbres AABB, regardons à quoi ressemble le squelette de l'algorithme de test.

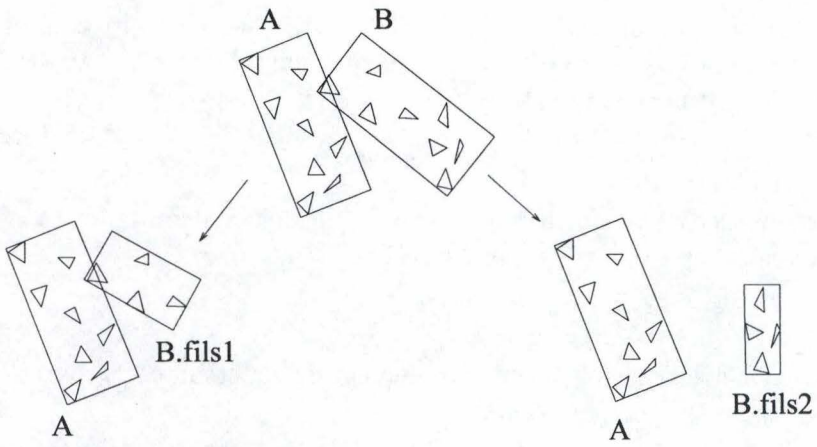


FIG. 4.26 – Deux tests supplémentaires sont nécessaires lorsque deux OBB se chevauchent

```
test(A, B)
{
  if (A chevauche B)
  {
    if (A.feuille && B.feuille)
    {
      if (A.triangle chevauche B.triangle)
      {
        ajouter_la_paire_a_la_liste_resultat;
        return;
      }
    }
    if (B.feuille || (!A.feuille && (A.taille > B.taille)))
    {
      test(A->fils1, B);
      test(A->fils2, B);
    }
    else
    {
      test(A, B->fils1);
      test(A, B->fils2);
    }
  }
}
```

Cet algorithme n'est qu'une ébauche. En réalité, il y a certaines choses importantes qui doivent être présentes. En permanence, il faut maintenir le positionnement de la boîte de

Maintenant prenons C, un fils de B (figure 4.27 (b)). Chaque boîte, comme nous l'avons vu plus haut, stocke la transformation qui la positionne relativement au système de coordonnées de son parent, notons la (R_2, T_2),

$$p_B = R_2 p_C + T_2$$

Etant donné ces deux transformations, nous voudrions en déduire la transformation qui positionne C dans le repère de A, notons la (R', T'),

$$p_A = R' p_C + T'$$

Nous obtenons cette transformation simplement en substituant p_B par $R_2 p_C + T_2$ dans l'équation $p_A = R_1 p_B + T_1$, ce qui donne

$$p_A = R_1(R_2 p_C + T_2) + T_1 = R_1 R_2 p_C + R_1 T_2 + T_1$$

Donc,

$$\begin{cases} R' &= R_1 R_2 \\ T' &= R_1 T_2 + T_1 \end{cases}$$

Nous pouvons maintenant effectuer le test de chevauchement entre les boites A et C.

A présent, si nous prenons D un fils de A (figure 4.27 (a)) et que nous voulons exprimer un sommet de B relativement au repère de D, les calculs seront un peu plus compliqués. En effet, nous serons obligés d'effectuer une inversion de matrice avant la substitution. Nous avons

$$p_A = R_1 p_B + T_1$$
$$p_A = R_3 p_D + T_3$$

et nous voulons obtenir la transformation (R'', T'') tel que

$$p_D = R'' p_B + T''$$

Pour cela, exprimons p_D en fonction de p_A,

$$p_D = R_3^{-1}(p_A - T_3)$$

et substituons de cette expression p_A par sa valeur,

$$p_D = R_3^{-1}((R_1 p_B + T_1) - T_3) = R_3^{-1} R_1 p_B + R_3^{-1}(T_1 - T_3)$$

Donc,

$$\begin{cases} R'' &= R_3^{-1} R_1 \\ T'' &= R_3^{-1}(T_1 - T_3) \end{cases}$$

Nous pouvons maintenant aussi effectuer le test de chevauchement entre les boites D et B.

Concentrons nous non plus sur le test de chevauchement de deux arbres de boîtes limites mais plutôt sur le test de chevauchement de deux boîtes limites orientées. Ainsi que se cache-t-il derrière le code

la seconde hiérarchie, B, relatif à la boîte de la première hiérarchie, A. Nous réalisons cela avec une seule composition de transformations, chaque fois que nous descendons dans un fils. Nous avons cette transformation parce que le test de chevauchement entre deux boites est rendu beaucoup plus efficace lorsqu'une des boîtes est alignée aux axes et centrée à l'origine. En résumé, lorsque nous appelons la procédure de test, nous supposons que la boîte limite A est alignée aux axes et centrée à l'origine, et que la position de B est spécifiée relativement à A.

Cette alignement aux axes et centrage de la boîte A est assez simple. Notons R^A, T^A et R^B, T^B les positionnements de A et B respectivement par rapport à leurs axes de coordonnées locaux. En appliquant la rotation $(R^A)^{-1} = (R^A)^T$ et la translation $-T^A$, c'est-à-dire la transformation $-(R^A)^T T^A$, à la boite A, nous obtenons que $T^A = (0,0,0)$ et $R_1^A = (1,0,0)$, $R_2^A = (0,1,0)$, $R_3^A = (0,0,1)$. Pour la boîte B, nous remplaçons R^B par $(R^A)^T R^B$ et T^B par $(R^A)^T(T^B - T^A)$. Ceci peut être réalisé dans une phase de pré-calcul.

Examinons de plus près les transformations permettant de réaliser une correspondance entre les différents repères.

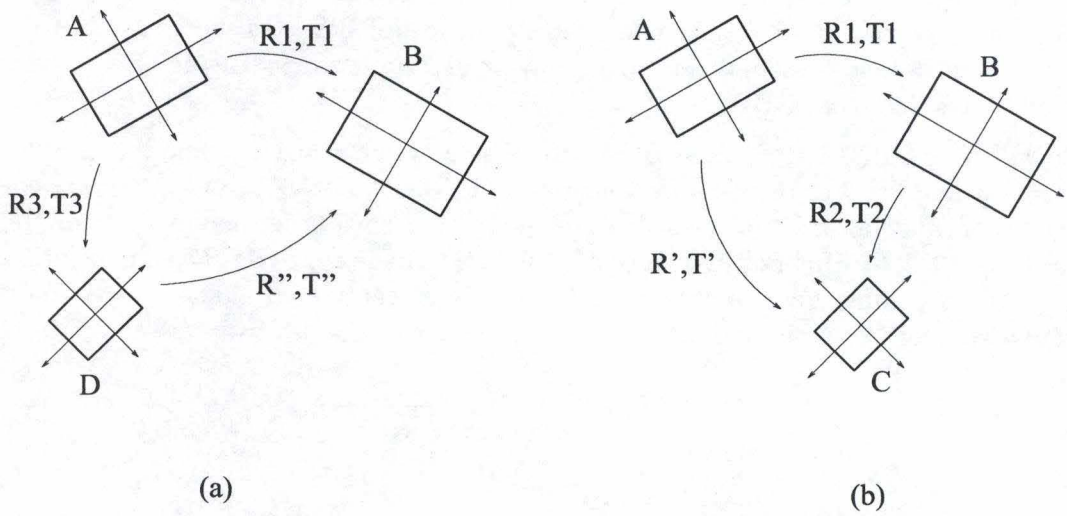


FIG. 4.27 – (a) Descente vers le fils D de A ou (b) C de B

Supposons que nous connaissons déjà la transformation (R_1, T_1) qui place la boîte B dans le système de coordonnées de A :

$$p_A = R_1 p_B + T_1$$

où p_B sont les coordonnées d'un sommet de B relatif aux axes de B, et p_A sont les coordonnées du même sommet mais mesurées relativement aux axes de A.

```
if (A chevauche B)
  { ...
```

L'algorithme "chevauche" renvoie vrai si les boîtes en entrées se chevauchent. Il est basé sur le théorème, en géométrie convexe, qui affirme que deux polytopes (tels que des boîtes) sont disjoints s'il existe un plan séparateur qui est soit parallèle à une face d'un des polytopes ou parallèle à une arête prise de chaque polytope.

Le concept de plan séparateur est en proche relation avec le concept d'axe séparateur. Un axe est une droite dans l'espace, et un axe est un axe séparateur si les boîtes, quand elles sont projetées sur lui, forment des intervalles qui ne se chevauchent pas. En définitive, un axe séparateur existe si et seulement si un plan séparateur existe, et le théorème ci-dessus a une forme duale impliquant des axes qui se formule : si deux polytopes sont disjoints, alors il existe un axe séparateur qui est soit perpendiculaire à une de leurs faces ou soit mutuellement perpendiculaire à une arête de chaque polytope, c'est-à-dire aligné avec le produit vectoriel de leur direction.

Dans le cas de deux boîtes, il y a au total six orientations de faces (nous ne faisons pas attention au signe des directions) et neuf produits vectoriels de directions d'arêtes, ce qui constitue en tout 15 directions candidates pour le test des axes séparateurs. Ainsi, le test sur 15 axes donnés est un test suffisant pour déterminer le statut de chevauchement de deux boîtes, ici, deux OBB.

Soient deux OBB A et B, avec B positionné relativement à A par une rotation R^B et une translation T^B. Les demi-dimensions de A et B sont notées a_i et b_i respectivement ($i = 1, 2, 3$). Nous notons également les axes de A et B sous la forme des vecteurs unités A^i et B^i ($i = 1, 2, 3$). Une petite remarque peut être faite à propos de R^B, si nous utilisons les axes de A comme base, n'oublions pas que A est centrée et alignée au système de coordonnées, alors les trois colonnes de R^B sont les même que les trois vecteurs B^i.

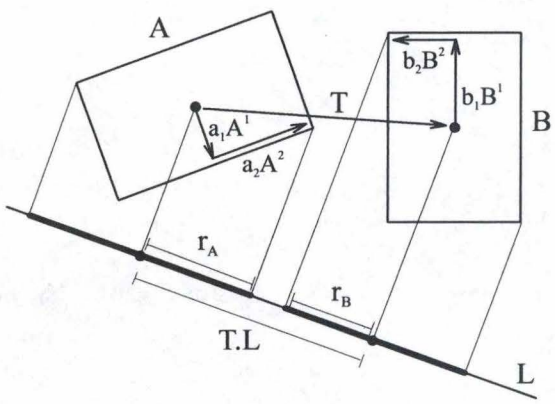


FIG. 4.28 – L est l'axe séparateur des OBB A et B parce que A et B produisent des intervalles disjoints une fois projetés sur L

Le test pour chaque direction L (vecteur unité) est implémenté comme suit :

1. calculer le rayon des intervalles résultant de la projection des boîtes sur l'axe. Ceci est calculer pour chaque boîte en projetant ses demi-dimensions sur l'axe et en sommant la longueur de leur image. Nous obtenons r_A et r_B,

$$\begin{cases} r_A &= \sum_{i=1}^{3} a_i \left| A^i \cdot L \right| \\ r_B &= \sum_{i=1}^{3} b_i \left| B^i \cdot L \right| \end{cases}$$

2. calculer la distance s entre la projection des centres des boîtes sur l'axe. Comme nous nous avons supposé que la boîte A est centrée à l'origine, la position du centre de B par rapport à A est en T. Donc,

$$s = \left| T \cdot L \right|$$

3. les intervalles sont disjoints si et seulement si $s > r_A + r_B$.

Il apparaît qu'un test pour un axe demande 54 opérations (7 produits scalaires, 7 valeurs absolues, 6 multiplications, 5 additions et une comparaison). Ce test se simplifie assez fort lorsque L est normal à une face de A ou B ou lorsque L est un produit vectoriel d'axes pris de chaque boîtes. De plus, rappelons nous que nous avons choisi de travailler dans un repère de coordonnées aligné avec A, donc $A^1 = [1,0,0]$, $A^2 = [0,1,0]$ et $A^3 = [0,0,1]$.

Prenons L une direction normale à une face de A, par exemple, $L = A^1$. Regardons comment la formule va se simplifier,

$$\left| T \cdot A^1 \right| > \sum_{i=1}^{3} a_i \left| A^i \cdot A^1 \right| + \sum_{i=1}^{3} b_i \left| B^i \cdot A^1 \right|$$

Premièrement, les produits vectoriels entre les axes de A se simplifient soit en 0, s'ils sont les mêmes, ou en 1, s'ils sont perpendiculaires. Deuxièmement, les produits vectoriels entre A^1 et un vecteur quelconque donne la première composante de ce vecteur. Donc, nous avons

$$\left| T_1 \right| > a_1 + b_1 \left| B_1^1 \right| + b_2 \left| B_2^1 \right| + b_3 \left| B_3^1 \right|$$

qui ne nécessite plus que 11 opérations, soit une économie de 43 opérations !

Maintenant, prenons L une direction normale à une face de B, par exemple, $L = B^1$. Nous obtenons finalement, ici $T \cdot B^1$ ne se simplifie pas,

$$\left| T_1 B_1^1 + T_2 B_2^1 + T_3 B_3^1 \right| > a_1 \left| B_1^1 \right| + a_2 \left| B_2^1 \right| + a_3 \left| B_3^1 \right| + b_1$$

qui nécessite 16 opérations, ici nous avons une économie de 38 opérations.

Pour les 6 tests précédents, nous avons besoin de $3*11+3*16 = 81$ opérations. Voyons à présent ce qu'il en est pour les directions L (il en reste 9) qui sont des produits vectoriels

de direction d'axe de A et B, par exemple, $L = A^1 \times B^1$. Pour ce cas, nous utiliserons la propriété suivante du produit vectoriel,

$$a \cdot (b \times c) = c \cdot (a \times b) = b \cdot (c \times a)$$

qui va nous permettre de réarranger les produits vectoriels, afin qu'ils ne s'appliquent plus qu'entre des vecteurs provenant des axes d'une même boîte. Ainsi, un produit vectoriel d'un vecteur avec lui-même donne le vecteur nul et un produit vectoriel d'un axe avec un autre axe donne l'axe manquant, par exemple, $B^1 \times B^3 = B^2$. Nous obtenons finalement,

$$\left|T_3 B_2^1 - T_2 B_3^1\right| > a_3 \left|B_2^1\right| - a_2 \left|B_3^1\right| + b_2 \left|A_3^1\right| - b_3 \left|A_2^1\right|$$

qui demande 16 opérations. Donc pour les 9 tests, il faut 144 opérations.

4.5.7 Les OBB contre les AABB et les sphères

Une des premières motivations qui nous poussent à utiliser les OBB est que, en vertu de leur orientation variable, elles peuvent englober la géométrie d'un objet avec plus de précision que les arbres AABB ou les arbres de sphères. Ainsi, peu de niveaux d'un arbre OBB ont besoin d'être traversés lors d'un test de collision entre des objets en fortes proximités.

Pour notre analyse, nous allons définir la *précision* τ, le *diamètre* d et la *proportion* ρ d'un volume limite en fonction de la géométrie qu'il englobe.

La *précision* τ d'un volume limite, B, en fonction de la géométrie qu'il englobe, G, est la *distance de Hausdorff dirigée*[6], en considérant B et G comme des ensembles de points,

$$\tau = \max_{b \in B} \min_{g \in G} \|b - g\|$$

Le *diamètre* d d'un volume limite est la distance maximum parmi toutes les paires de points enfermés dans le volume,

$$d = \max_{g,h \in G} \|h - g\|$$

La *proportion* ρ est donnée par le rapport

$$\rho = \frac{\tau}{d}$$

[6]Etant donné deux ensembles de points $A = \{a_1, ..., a_m\}$ et $B = \{b_1, ..., b_n\}$, la distance de *Hausdorff* est définie par

$$H(A, B) = \max(h(A, B), h(B, A))$$

où $h(A, B) = \max_{a \in A} \min_{b \in B} \|a - b\|$. La fonction $h(A, B)$ est appelée la distance de Hausdorff *dirigée* de A à B (cette fonction n'est pas symétrique et donc n'est pas une vraie distance). Elle détermine le point $a \in A$ qui est le plus éloigné de tous les points de B, et mesure la distance Euclidienne de a à son voisin le plus proche dans B.

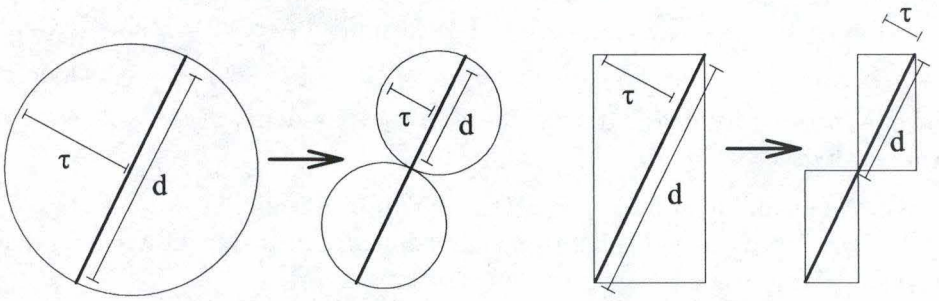


FIG. 4.29 – Les proportions ρ des volumes parents sont similaires à celles des fils lorsqu'ils englobent un objet dont la géométrie est presque plate

Lorsque les surfaces à englober ont une faible courbure, les arbres AABB et les arbres de sphères forment des hiérarchies avec des proportions fixes, c'est-à-dire que la proportion ρ d'un noeud de la hiérarchie est identique à la proportion de ces fils (figure 4.29). Si la géométrie de l'objet englobé est presque plate, alors les fils ont une forme similaire à leurs parents, mais plus petit. Dans la figure 4.29, d et τ sont divisés par deux lorsque l'on passe des parents aux fils, ainsi $\rho = \frac{\tau}{d}$ est approximativement le même pour un parents et ses fils.

Il faut noter que la proportion ρ pour les AABB est très dépendante de l'orientation de l'objet géométrique englobé. Si l'objet est aligné convenablement, alors la proportion ρ peut être proche de 0, par contre si l'objet est mal aligné alors cette même proportion peut être proche de 1.

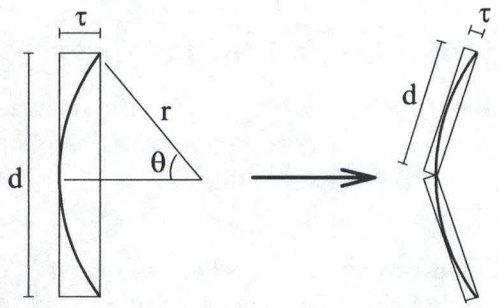


FIG. 4.30 – La proportion ρ des volumes fils est la moitié de celle du parent lorsqu'ils englobent une surface de faible courbure

Puisque les OBB s'alignent sur la géométrie de l'objet, la proportion ρ ne dépend plus de l'orientation de l'objet englobé mais plutôt de la courbure locale de celui-ci. Supposons que la géométrie de l'objet englobé à une courbure faible, comme la surface d'une large sphère. Dans la figure 4.30, nous considérons une courbe plane avec un rayon de courbure

fixe r et englobé par une OBB. Nous avons $d = 2r\sin\theta$ et $\tau = r - r\cos\theta$. Nous pouvons en déduire $\sin\theta = \frac{d}{2r}$ et ainsi calculer $\cos\theta$ par la formule $1 - \cos\theta = 2\sin^2\frac{\theta}{2}$. Cela donne $\cos\theta = 1 - \frac{d^2}{8r^2}$. Nous obtenons finalement que $\tau = r - r\left(1 - \frac{d^2}{8r^2}\right) = \frac{d^2}{8r}$. Donc τ a une dépendance quadratique avec d. Lorsque d est divisé par deux, τ l'est par quatre et la proportion ρ est divisée par deux.

Nous en concluons que lorsque la géométrie de l'objet à englober à une faible courbure, les arbres AABB et les arbres de sphères ont un τ linéairement dépendant à d, tandis que pour les arbres OBB, τ a une dépendance quadratique à d.

Supposons à présent que nous aillons besoin de N volumes limites de même taille pour couvrir un morceau d'une surface avec une aire A, et exigeons que chaque volume limite couvre une portion de l'aire de la surface de l'ordre de $O(A/N)$. Par conséquent, pour ces volumes, $d = O(\sqrt{A/N})$. Pour les AABB et les sphères, τ dépend linéairement de d, donc $\tau = O(\sqrt{A/N})$. Pour les OBB, la dépendance quadratique avec d donne $\tau = O(A/N)$. Donc pour couvrir un morceau de surface avec des volumes limites étant donné une certaine précision, si les OBB nécessitent $O(m)$ volumes limites alors les AABB et les sphères devraient nécessiter $O(m^2)$ volumes limites. Dans la figure 4.31, nous remarquons intuitivement que puisque les arbres OBB convergent plus vite vers la forme de l'objet qu'ils englobent que les AABB, alors le nombre de volumes limites nécessaires pour atteindre une certaine précision de recouvrement sera beaucoup plus petit pour les OBB que les AABB.

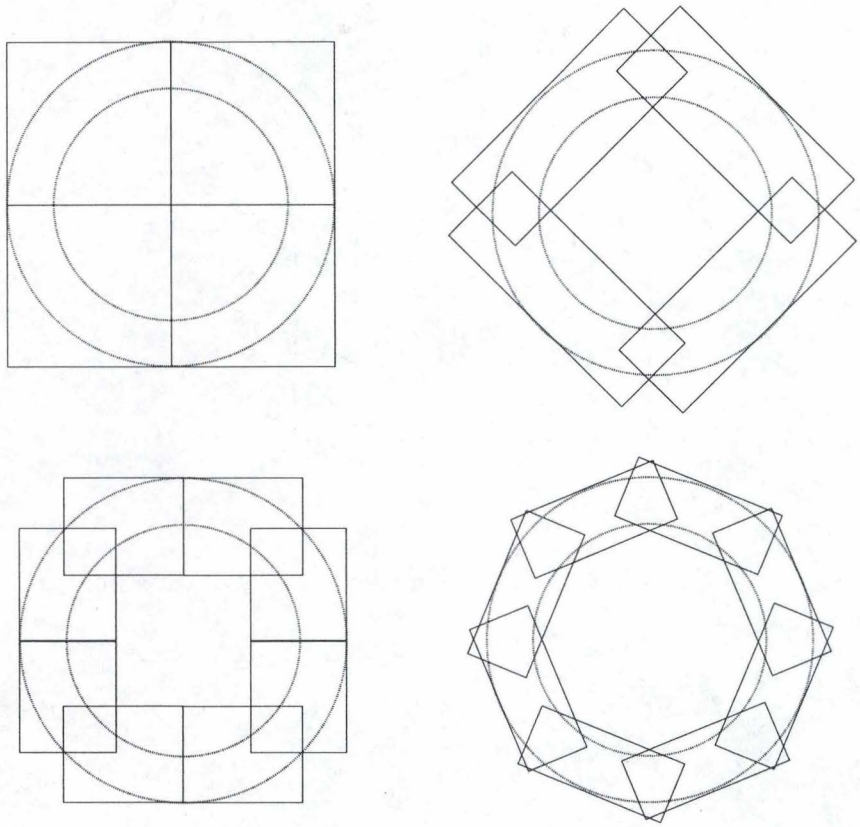
La plupart des scénarios de contact ne nécessitent pas de traverser tous les noeuds d'une profondeur donnée des deux arbres, mais cela se passe quand deux surfaces entrent en forte proximité parallèle l'une l'autre, c'est-à-dire lorsque beaucoup de points d'une surface sont proches de certains points de l'autre surface. C'est très commun dans le prototypage virtuel où des parties de machine sont testées.

4.5.8 k-DOP - polytope à orientations discrètes

Nous allons nous intéresser maintenant à une famille de volumes limites qui sont des polytopes (convexes) dont les faces sont déterminées par des plans parallèles[7] dont les normales extérieurs proviennent d'un petit ensemble fixé de k orientations. Nous appellerons de tels polytopes des *polytopes à orientations discrètes* ou "k-DOP".

Le k-DOP a été conçu pour combler l'insuffisance de la AABB qui était une pauvre approximation de l'ensemble de primitives qu'elle englobait, laissant de larges coins vides. Dans la figure 4.32, 6 plans sont nécessaire pour construire la AABB de l'objet, ensuite les coins vides sont tronqués à l'aide de 8 autres plans et le résultat obtenu est un 14-DOP. L'utilisation de k-DOP, avec des valeurs de k élevées, permet au volume limite d'approximer l'enveloppe convexe plus précisément. Bien entendu, l'amélioration de l'approximation, qui a tendance à diminuer N_v, N_p et N_u, favorise l'augmentation des coûts C_v (puisque $C_v = O(k)$) et C_u (puisque $C_u = O(k^2)$). Le choix du paramètre k nous donne un moyen

[7]dans la littérature anglaise ces plans sont appelés "slabs"



(a) (b)

FIG. 4.31 – (a)AABB contre (b)OBB : les niveaux 3 et 4 de l'approximation d'un tore. Les OBB convergent plus vite vers la forme du tore que les AABB.

de trouver un compromis entre la mauvaise approximation des sphères limites et AABB, et les coûts élevés des tests de chevauchement et de mise à jour associés aux OBB. De plus, les vecteurs normaux sont choisis de telle manière que leurs coordonnées soient des entiers pris dans l'ensemble $\{-1, 0, 1\}$, cela a comme avantage qu'aucunes multiplications n'est nécessaire lorsque nous effectuons des calculs avec eux.

Les plus utilisés sont les 6-DOPs, 14-DOPs, 18-DOPs et 26-DOPs. Les 6-DOPs sont en fait une classe de volumes limites que nous avons déjà rencontré, il s'agit des AABB dont les vecteurs d'orientation sont déterminés par les axes de coordonnées pris dans le sens positif et négatif, en effet, il suffit de trouver les valeurs minimums et maximums des coordonnées des sommets des primitives dans les directions de x, y et z. Pour la suite, nous n'envisagerons plus que les 14-DOPs dont le traitement est identique à tout autre k-DOP.

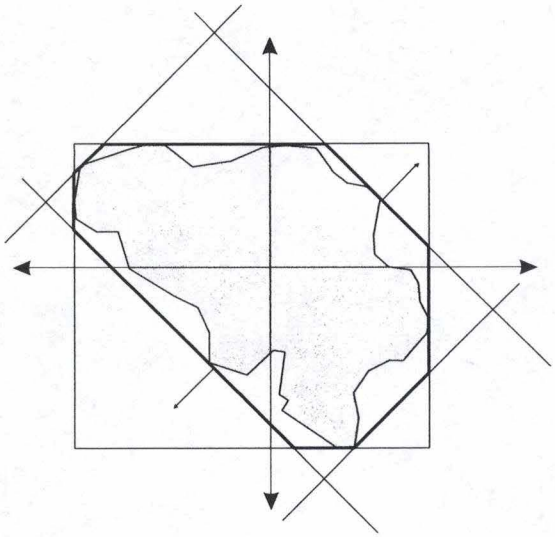


FIG. 4.32 – Une vue bidimensionnelle du 14-DOP d'un objet.

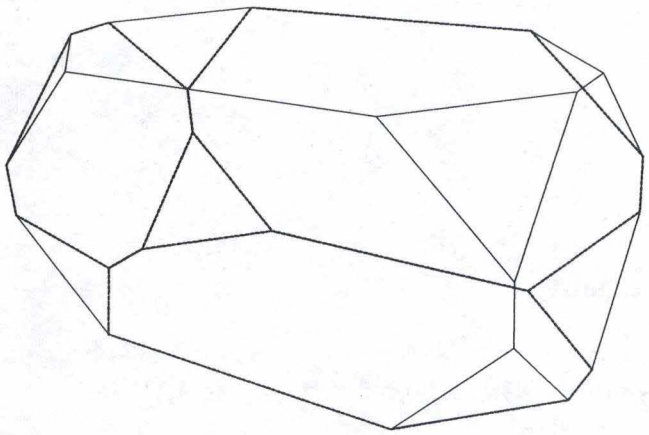


FIG. 4.33 – Une vue tridimensionnelle d'un 14-DOP

Construction d'un polytope à orientations discrètes

Un k-DOP requiert pour son stockage en mémoire k scalaires. Donc pour le cas que nous analysons, le 14-DOP nécessite 14 scalaires pour son stockage.

La construction d'un k-DOP est extrêmement simple. Pour un 14-DOP, il suffit de trouver le minimum et le maximum des coordonnées des sommets des primitives d'un objet le long de chacun des 7 axes, définis par les vecteurs $(1,0,0)$, $(0,1,0)$, $(0,0,1)$, $(1,1,1)$, $(1,-1,1)$, $(1,1,-1)$ et $(1,-1,-1)$. De manière plus précise,

Soit S, un ensemble de n primitives géométriques (triangles). Chaque primitive est

constituée de 3 sommets p^i, q^i et r^i ($i \in \{1, .., n\}$). Chacun de ces sommets est composé de trois coordonnées (x_i, y_i, z_i) ($i \in \{1, .., 3n\}$). Notons les 14 scalaires définissant un 14-DOP $v_1, v_2, ..., v_{14}$. Ces v_j ($j \in \{1, .., 14\}$) doivent vérifier les inégalités suivantes : $\forall i \in \{1, .., 3n\}$,

$$\begin{array}{rcl@{\qquad}rcl} x_i & \geq & v_1 & x_i & \leq & v_8 \\ y_i & \geq & v_2 & y_i & \leq & v_9 \\ z_i & \geq & v_3 & z_i & \leq & v_{10} \\ x_i + y_i + z_i & \geq & v_4 & x_i + y_i + z_i & \leq & v_{11} \\ x_i - y_i + z_i & \geq & v_5 & x_i - y_i + z_i & \leq & v_{12} \\ x_i + y_i - z_i & \geq & v_6 & x_i + y_i - z_i & \leq & v_{13} \\ x_i - y_i - z_i & \geq & v_7 & x_i - y_i - z_i & \leq & v_{14} \end{array}$$

L'algorithme qui permet de calculer les v_j est un algorithme incrémental. Le premier pas de l'algorithme initialise tout les v_j en leur donnant la valeur des membres gauches des inégalités, leur étant associé, évalués avec le premier sommet (x_1, y_1, z_1). Dans le second pas, l'algorithme évalue les membres gauches des 14 inégalités avec le second sommet (x_2, y_2, z_2) et ensuite les testes. Si une inégalité n'est pas respectée, prenons par exemple l'inégalité associée à v_4, alors l'algorithme la corrige en remplaçant la valeur du v_j erronée par la valeur du membre gauche, ici pour notre exemple $v_4 = x_2 + y_2 + z_2$. L'algorithme continue ainsi de suite, en prenant modèle sur le second pas, avec les $3n - 2$ sommets restants. La complexité algorithmique de cet algorithme est $O(n)$.

Nous n'aborderons pas ici la construction de l'arbre car celle-ci est réalisée de la même façon que pour les arbres AABB.

Test d'intersection

Lorsque qu'une demande de détection de collision est traitée, la fonction la plus fréquemment appelée est celle qui teste si deux k-DOPs se chevauchent. Le coût de cette opération est noté C_v. Il faut se rappeler que tous les k-DOPs sont définis par le même ensemble fixé de k directions pour un k particulier. Donc un k-DOP est complètement déterminé par $k/2$ intervalles décrivant son étendue le long des $k/2$ directions. Deux k-DOPs A et B ne se chevauchent pas si au moins un des $k/2$ intervalles de A n'interfère pas avec l'intervalle de B correspondant. Si les k-DOPs A et B se chevauchent le long de toutes les $k/2$ directions, alors nous concluons qu'ils sont en collision.

Pour les cas de deux 14-DOPs A et B, il faut vérifier qu'au moins un des 7 intervalles de A n'interfère pas avec les intervalles correspondants de B (figure 4.34). Soient v_j et w_j ($j \in \{1, .., 14\}$) les 28 scalaires définissant les intervalles de A et B respectivement. Si une des inégalités suivantes est vérifiée alors les k-DOPs A et B sont disjoints.

$$
\begin{array}{llll}
w_1 & > v_8 & v_1 & > w_8 \\
w_2 & > v_9 & v_2 & > w_9 \\
w_3 & > v_{10} & v_3 & > w_{10} \\
w_4 & > v_{11} & v_4 & > w_{11} \\
w_5 & > v_{12} & v_5 & > w_{12} \\
w_6 & > v_{13} & v_6 & > w_{13} \\
w_7 & > v_{14} & v_7 & > w_{14}
\end{array}
$$

Nous n'avons besoin pour tester un chevauchement de deux k-DOPs que de k opérations de comparaison.

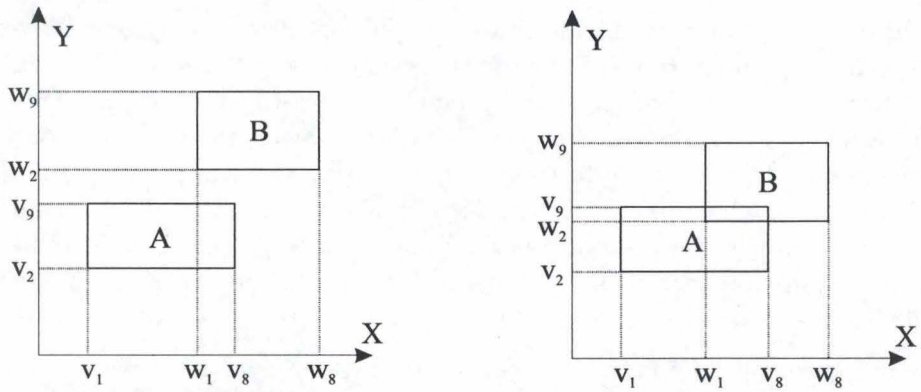


FIG. 4.34 – (a) les 6-DOPs A et B ne sont pas en collision ($w_2 > v_9$) ; (b) A et B sont en collision

Le test d'intersection entre deux arbres de k-DOPs, approximant chacun un objet, se fait en testant récursivement chaque paire de noeud (comme pour les arbres AABB) jusqu'à arriver au niveau des feuilles qui contiennent les primitives géométriques. Uniquement les noeuds pour lesquels les k-DOPs associées sont en intersection sont parcourus plus en avant. Ici, puisque tous les k-DOPs sont issus d'un même ensemble de k directions, les changements de repère d'un arbre vers un autre ne sont plus nécessaires. Cela accélère grandement le processus de détection de collision.

4.5.9 Les OBB contre les k-DOPs

Nous allons ici tenter de cerner les avantages et limitations des deux principaux types de hiérarchies de volumes limites pour la détection de collision entre des objets polygonaux. Nous allons nous baser sur les critères de conception vu précédemment au point 4.5.1.

Les principaux avantages de l'utilisation de OBB dans une hiérarchie sont qu'ils :

1. permettent d'atteindre une approximation plus fine d'un objet. Les dimensions et orientations d'un ensemble de primitives géométriques peuvent être approximées par trois vecteurs propres orthogonaux provenant d'une matrice de covariance.
2. assurent une construction efficace de l'arbre

D'un autre côté, les principaux problèmes sont que :

1. le test de chevauchement entre deux OBB est coûteux. Malgré l'utilisation du théorème des axes séparateurs, les tests restent plus lents que pour les autres volumes limites tel que les sphères ou les AABB.
2. une demande de détection de collision entre deux arbres de OBB nécessite le calcul de matrice de transformation entre les divers systèmes de coordonnées (N_u est élevé). Puisque les OBB ont des orientations arbitraires, des multiplications de matrices 4x4 doivent être calculées avant chaque demande. En d'autres mots, $N_u = N_v$. Le coût C_u est aussi élevé que la moitié du coût C_v.

Nous pouvons résumer les principaux avantages des k-DOPs comme ceci.

1. les k-DOPs établissent un compromis entre la faible approximation des sphères, AABB et l'approximation maximum des enveloppes convexes. Puisque les orientations des polytopes sont prédéfinies, une plus grande finesse d'approximation est atteinte en augmentant le nombre de plans parallèles (k).
2. la détection de collision est très rapide entre deux k-DOPs grâce à une simple vérification d'intervalle. D'un autre côté, C_v est directement proportionnel à k.
3. N_u est fort réduit puisque les orientations des plans parallèles sont fixées. En effet, le k-DOP d'un noeud dans une hiérarchie n'a besoin d'être transformé qu'une fois, lors de la première visite, vers le k-DOP de la seconde hiérarchie. Il en résulte donc que $N_u << N_c$.

Les principaux problèmes sont :

1. il est difficile de déterminer le k optimal donnant une bonne approximation de l'objet et des tests de chevauchement rapides. En général, k varie de 6 jusqu'à 26, mais $k = 14$ et $k = 18$ restent les meilleurs choix.
2. le coût C_u est élevé, il croit linéairement avec k. En moyenne C_u est dix fois plus grand que C_v.

En regardant ces deux comparaisons, nous remarquons qu'il serait intéressant de concevoir un volume limite qui réalise une fine approximation, donne un test de chevauchement rapide, nécessite un petit nombre de mise à jour de noeuds (N_u) et un faible coût de mise à jour (C_u).

Certains chercheurs [He99] ont proposé un nouveau volume limite qu'ils ont appelé "QuOSPOs", c'est-à-dire Quantized Orientation Slabs with Primary Orientations. Leur but est de combiner et accroître les avantages des OBB et k-DOPs.

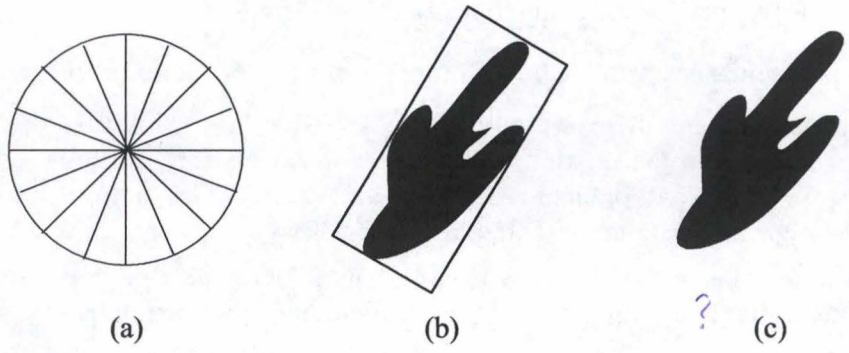


FIG. 4.35 – Approximation d'un objet avec un QuOSPO. (a) Quantized Orientation (codée sur 4 bits définissant 16 orientations), (b) approximation par une OBB, (c) approximation par un QuOSPO.

Chapitre 5

La détection de collision entre paire d'objets (approche matérielle)

Jusqu'à présent, aucun des algorithmes que nous avons étudié n'exploitaient les capacités du matériel graphique (hardware). Certaines cartes graphiques ont des commandes de gestion de caméra câblées dans leur processeur, notamment les commandes OpenGL.

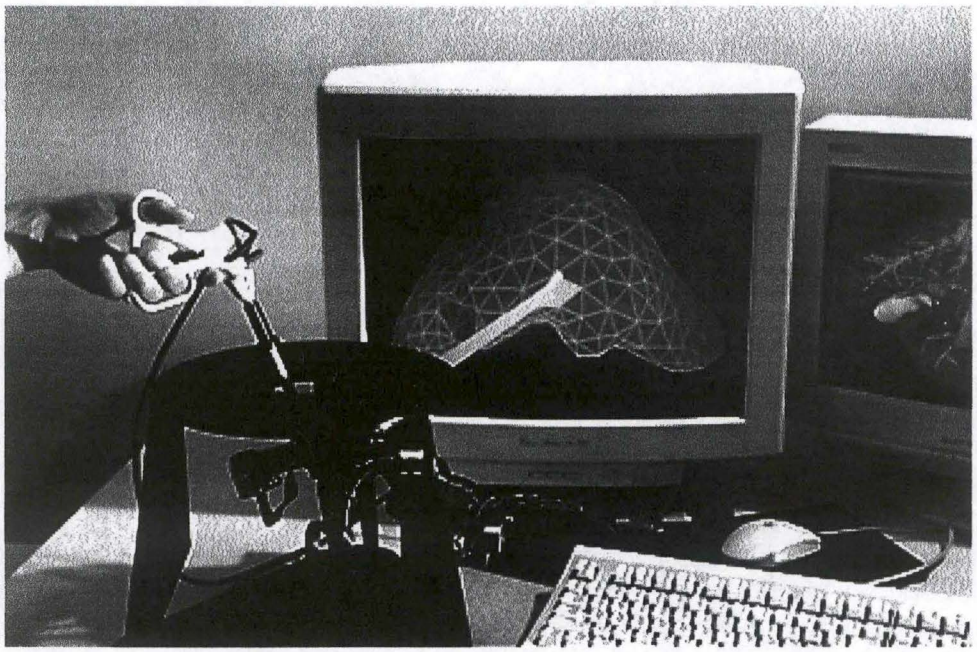
Nous allons étudier un exemple d'utilisation de ces propriétés matérielles dans une application de chirurgie virtuelle (simulation de chirurgie).

5.1 Introduction

La laparoscopie a actuellement un énorme succès auprès des chirurgiens, notamment puisqu'elle réduit fortement le temps d'opération et la morbidité. En effet, la laparoscopie consiste à introduire par une petite incision ouverte, sur l'abdomen du patient, plusieurs outils (notamment un laparoscope) et une fibre optique supportant une micro-caméra. Le chirurgien, qui doit couper et enlever une région pathologique d'un organe, visualise l'opération uniquement à travers un écran. Apprendre à bien coordonner les mouvements des outils dans ces conditions est une tâche extrêmement difficile.

Le but d'un simulateur de chirurgie est d'offrir un outil permettant aux chirurgiens de s'entraîner sur un patient virtuel, ainsi cela permet d'éviter de s'entraîner sur des animaux vivants ou bien des cadavres humains.

Dans le contexte de la chirurgie virtuelle, le problème de la détection de collision est accru par la non-convexité de la plupart des organes et aussi par le fait que les organes se déforment au cours du temps. Donc passer du temps pour précalculer une hiérarchie de volumes limites sur un organe ne semble pas adéquat, puisque ce calcul devra être refait après chaque tranche de temps écoulée. Nous allons plutôt exploiter les capacités du matériel graphique, ceci au vu des caractéristiques de notre problème de chirurgie laparoscopique.



Référence 99 078 ©INRIA/Photo : A.Eidelman

FIG. 5.1 – Simulation de chirurgie hépatique

5.2 La détection de collision avec le matériel graphique

L'idée de base de la méthode est de définir un volume de visibilité de la caméra correspondant à la forme de l'outil (ou bien au volume couvert par l'outil entre deux tranches de temps successives). Nous utilisons le matériel pour afficher l'objet principal (organe) relativement à la caméra que nous venons de définir. Si rien n'est visible, alors nous pouvons en déduire qu'il n'y a pas de collision entre l'organe et l'outil. Sinon nous pouvons obtenir la partie de l'objet (les primitives géométriques) que l'outil touche.

Deux problèmes apparaissent :

- la forme de l'outil n'est pas aussi simple que le volume de visibilité et
- nous ne voulons pas avoir l'image, mais nous avons besoin des informations. Plus précisément, nous voudrions savoir quelles faces de l'objet sont impliquées dans la collision et en quelles coordonnées.

Les caméras virtuelles les plus couramment utilisées sont la caméra orthographique et la caméra en perspective (celle que nous avons vu dans la section 1.5 du chapitre 1). Dans les deux cas, le volume de visibilité est un hexaèdre, respectivement une boîte et une pyramide tronquée (figrure 5.2).

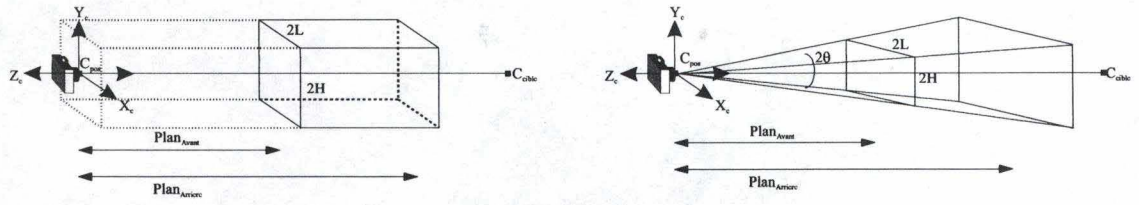


FIG. 5.2 – (a) caméra orthographique (le volume de visibilité est une boîte) et (b) caméra en perspective (le volume de visibilité est une pyramide tronquée)

5.2.1 Détection statique

Le laparoscope peut être vu comme un cylindre de section constante s et de longueur variable, puisque l'utilisateur peut le tirer ou le pousser plus ou moins profondément dans l'abdomen du patient. Nous allons appeler p_0 le point fixe où l'outil entre dans l'abdomen (p_0 est situé au centre de la petite incision par laquelle passe le laparoscope), et p l'extrémité de l'outil.

La détection d'une collision entre le laparoscope et la représentation polygonale de l'organe peut être facilement testée en associant une caméra orthographique à l'outil. En effet, la caméra est positionnée au point p_0 et la cible au point p. Le plan avant et arrière sont fixés respectivement à 0 et $\|p - p_0\|$. La section du laparoscope est prise en compte en donnant à H et L la valeur s.

Si nous voyons un ou plusieurs polygones avec cette caméra, alors cela signifie que le laparoscope est en collision avec un ou plusieurs polygones appartenant à la représentation de l'organe (figure 5.3).

5.2.2 Détection avec un mouvement

La solution précédente ne teste la collision uniquement à un moment donné et dans une position statique de l'outil et de l'organe. À présent, si l'utilisateur fait un mouvement rapide, l'outil peut traverser un petit morceau de l'organe sans qu'aucune collision ne soit détectée. Pour résoudre cette insuffisance, nous allons devoir prendre en compte le volume balayé par l'outil durant une certaine période de temps (pour des raisons de simplicité, nous supposons que l'organe n'a pas subi de déformations pendant cette tranche de temps).

Le laparoscope se déplace à l'intérieur de l'abdomen du patient à partir du point p_0 et pénètre plus ou moins profondément à travers ce point, donc sa longueur varie au cours du temps. Nous allons supposer que l'extrémité de l'outil de déplace de manière rectiligne de p vers p' (figure 5.4(a)). Puisque l'outil peut être vu comme un cylindre de rayon s, le volume couvert par l'outil durant une tranche de temps est approximé par un hexaèdre (figure 5.4(b)).

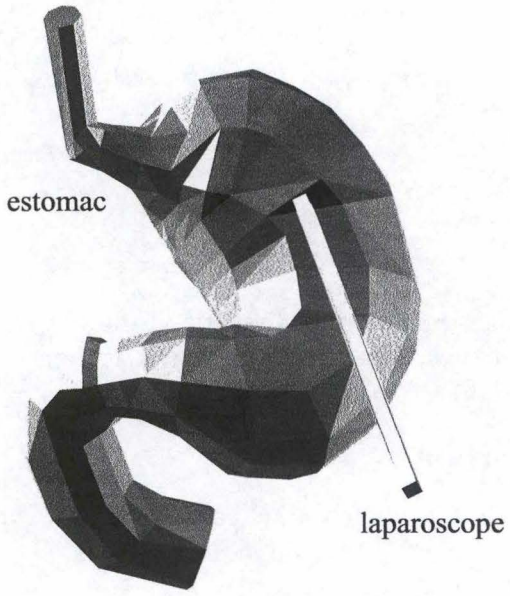


FIG. 5.3 – Détection d'une collision entre un objet graphique représentant un estomac et un outil statique

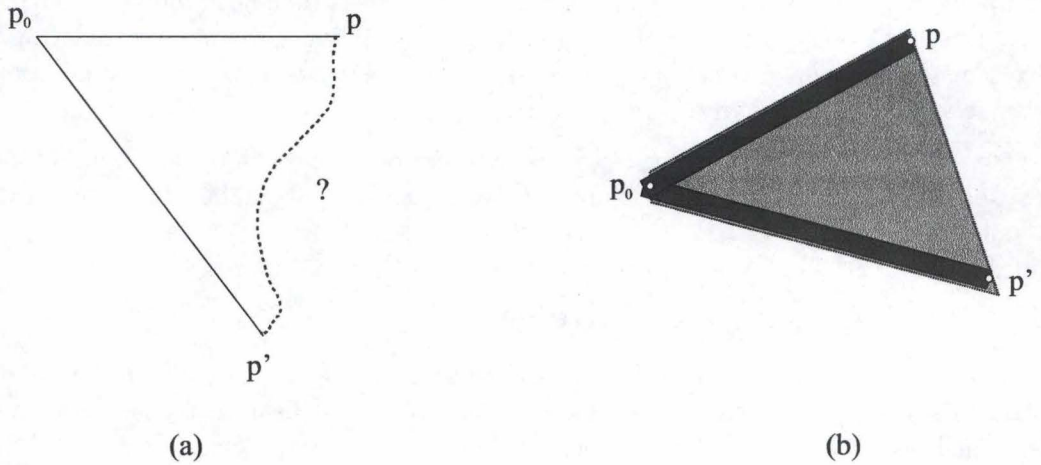


FIG. 5.4 – (a) le mouvement du laparoscope entre deux pas de temps et (b) le volume balayé par le laparoscope durant une tranche de temps

La détection de collision est réalisée en utilisant une caméra en perspective. Nous allons utiliser un volume de visibilité dont le cône suit les segments p_0p et p_0p'. Les axes de la caméra sont définis par la direction pp' pour l'axe X_c, par la direction $p_0p' \times p_0p$ pour l'axe Y_c et par la direction $(p_0p' \times p_0p) \times pp'$ pour l'axe Z_c (figure 5.5). Pour ce type de caméra, nous sommes obligés de rajouter deux plans de découpe afin de limiter verticalement (le

long de l'axe Y_c) l'étendue du volume à une hauteur de $2s$ (figure 5.6).

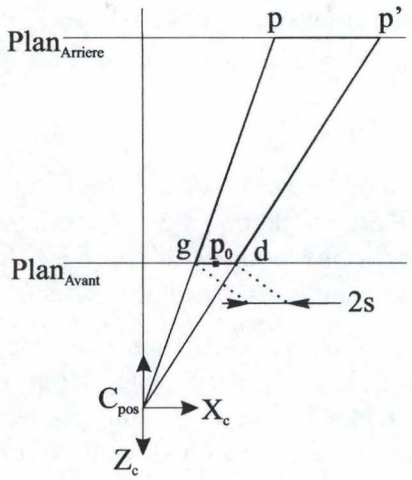


FIG. 5.5 – Vue du plan X_cZ_c de la caméra en perspective

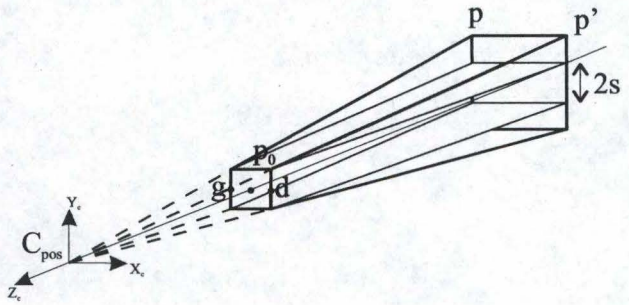


FIG. 5.6 – Réduction du volume de visibilité avec deux plans de découpe

La position de la caméra C_{pos} est calculée à partir des points p_0, p et p'. Soit u le vecteur unité dans la direction de l'axe X_c, $u = \frac{pp'}{\|pp'\|}$. Nous l'utilisons pour fixer les limites gauche et droite, notée respectivement g et d, du volume de visibilité sur le plan avant de découpe. Donc, $g = p_0 - su$ et $d = p_0 + su$. Grâce aux deux triangles semblables $C_{pos}pp'$ et $C_{pos}gd$, nous avons les égalités suivantes :

$$\frac{\|C_{pos}g\|}{\|C_{pos}p\|} = \frac{\|C_{pos}d\|}{\|C_{pos}p'\|} = \frac{\|gd\|}{\|pp'\|}$$

Ainsi, nous obtenons la position

$$C_{pos} = g - \frac{\|gd\|}{\|pp'\| - \|gd\|} gp$$

Pour ce qui concerne le positionnement des plans avant, arrière et autres, nous pouvons nous référer à [LCF99] car il est nécessaire de faire appel à des fonctions de la librairie graphique OpenGL spécialisées qui ne sont pas l'objet de ce chapitre.

5.3 Conclusion

Les travaux de [LCF99] ont montré que cette méthode était environ cent fois plus rapide que la méthode utilisant une hiérarchie de boîtes limites orientées. Ceci est du en partie au fait qu'aucun précalcul n'est nécessaire pour cette approche. Elle s'applique donc très bien à des scènes dynamiques où les objets se déplacent et se déforment au cours du temps.

Nous pouvons étendre l'utilisation de cette approche à des configurations de collisions plus complexes entre deux objets, l'un étant très simple (eg. la boîte limite d'un objet) et l'autre très complexe. Nous pourrions l'utiliser dans une application interactive de sculpture (figure 5.7), où l'utilisateur manipulerait un outil rigide pour éditer un objet déformable tridimensionnel.

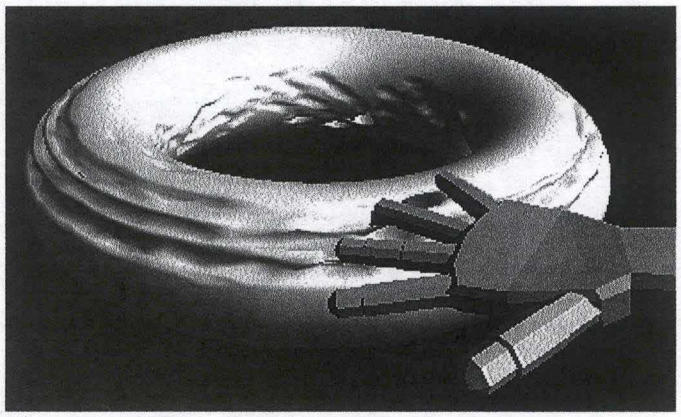


FIG. 5.7 – Sculpture virtuelle

Chapitre 6

Les extensions de la détection de collision

6.1 La détection de collision entre plusieurs objets

Un environnement virtuel de grande taille, tel qu'un "walkthrough"[1] (figure 6.1), créé par un ordinateur, est composé d'objets réels et virtuels. Ces objets doivent paraître solides et être manipulables pour que ce monde virtuel paraisse vraisemblable. C'est là qu'interviennent les algorithmes de détection de collision. Cependant, il peut y avoir des milliers d'objets dans le monde virtuel, et donc une approche par la force (brute force) qui teste toutes les paires d'objets possibles deux à deux n'est pas acceptable.

Nous allons voir deux procédés qui sont capables de réduire les $O(n^2)$ interactions possibles entre n objets en mouvement en $O(n+m)$ où m est le nombre d'objets en forte proximité l'un de l'autre, c'est-à-dire, deux objets sont en forte proximité si leurs boîtes limites à axes alignés associées se chevauchent.

6.1.1 Introduction

Les environnements virtuels contiennent aussi bien des objets stationnaires que des objets en mouvements. Par exemple, un utilisateur peut se déplacer à l'intérieur d'une construction dans laquelle les tables, chaises, etc. restent stationnaires.

Soit N le nombre d'objets en mouvement et M le nombre d'objets stationnaires. Chacun des N objets en mouvement peut entrer en collision avec d'autres objets en mouvement, aussi bien qu'avec des objets stationnaires. Tester en permanence, à chaque tranche de temps, les $\binom{N}{2} + NM$ paires d'objets devient vite très coûteux en temps lorsque N et M sont élevés. Pour arriver à un niveau interactif, nous devons réduire ce nombre avant d'effectuer la détection de collision entre paires d'objets. Nous allons utiliser la propriété

[1] Un "Walkthrough" est une application qui crée un environnement virtuel, tel que l'intérieur d'une maison, dans lequel un utilisateur peut se promener librement et interagir avec les objets l'entourant.

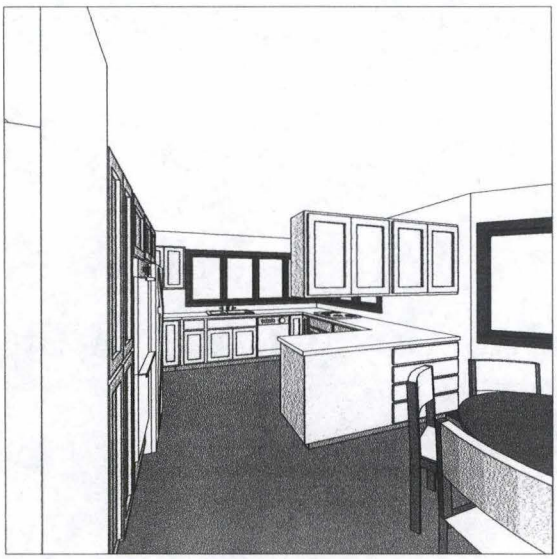


FIG. 6.1 – Walkthrough à l'intérieur d'une cuisine

de cohérence qui va nous permettre de diminuer fortement le nombre de tests entre paire d'objets à effectuer à chaque étape de temps.

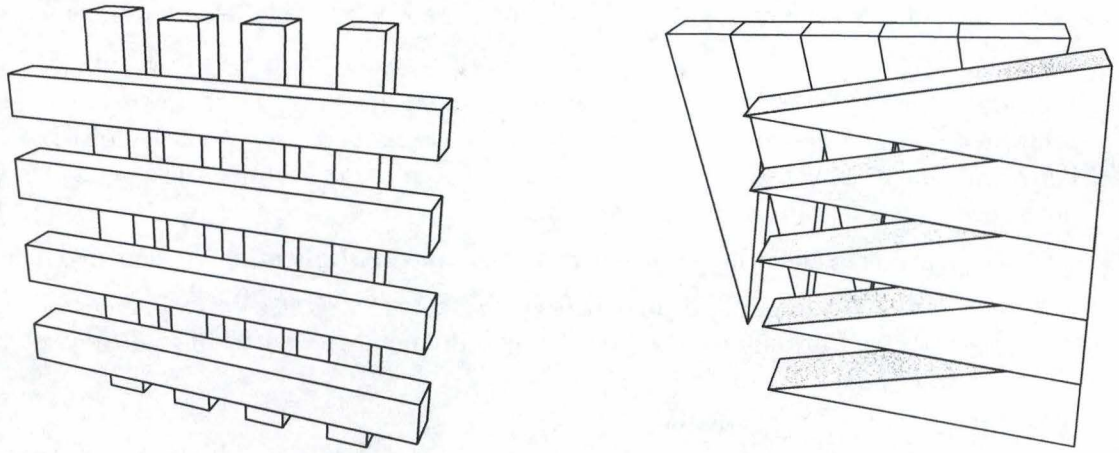
La cohérence temporelle est la propriété que l'état de l'application ne change pas significativement entre des tranches de temps successives. Les objets se déplacent légèrement d'image en image. Ce léger mouvement des objets se traduit également en cohérence géométrique puisque leur géométrie, définie par les coordonnées des sommets, change faiblement entre les images successives de l'animation. L'idée sous-jacente est que les tranches de temps sont suffisamment petites pour que les objets ne se déplacent pas sur de larges distances entre les images d'une animation.

Pour une configuration de n objets, le pire temps d'exécution pour n'importe quel algorithme de détection de collision est $O(n^2)$. Par exemple, prenons la figure 6.2 (a) où toutes les paires d'objets sont en collision. Une situation similaire peut se produire pour toutes les paires de composants de deux objets en collision, comme nous pouvons le voir sur la figure 6.2 (b). Cependant, nous pouvons nous rassurer car ces cas se rencontrent très rarement dans des environnements virtuels.

Nous allons analyser deux approches différentes, toutes deux travaillant sur les boîtes limites des objets de l'environnement virtuel, pour éliminer les tests de détection de collision sur des paires d'objets fort éloignés l'un de l'autre.

La première, utilisée dans des packages du domaine publique nommés "V-COLLIDE" et "I-COLLIDE" [2], est appelée "Sweep and Prune". Elle est basée sur l'utilisation d'axes

[2]réalisés par Min C. Lin et Dinesh Monacha à l'Université de Caroline du Nord



(a) (b)

FIG. 6.2 – Les pires scénarios pour la détection de collision

séparateurs. La deuxième approche, utilisée dans un package nommé "Q-COLLIDE"[3], est basée sur l'utilisation de plans séparateurs pour rapidement éliminer les paires d'objets fortement éloignées.

6.1.2 Approche 1 : Sweep and Prune

Trier les boîtes limites englobant les objets est la clé de l'approche "Sweep and Prune". Nous allons trier ces volumes limites pour déterminer quelles paires se chevauchent. Nous n'aurons plus qu'à effectuer une détection de collision sur ces paires restantes par les techniques vues dans le chapitre 4 page 36.

Toutefois, il n'est pas intuitivement évidant de savoir comment trier des boîtes limites dans un espace à trois dimensions pour déterminer les chevauchements. Nous allons utiliser une approche de réduction de la dimension. Si deux boîtes limites sont en collision dans un espace tridimensionnel, alors leurs projections orthogonales aux axes X, Y et Z se chevauchent.

Nous allons considérer pour cette approche deux types de boîtes limites à axes alignés : des cubes limites à taille fixe et des boîtes limites rectangulaires dynamiques (à taille variable).

- Cubes limites à taille fixe :
  Il faut calculer la dimension du cube pour qu'elle soit suffisamment large pour contenir un objet qui puisse prendre n'importe quelle orientation. Ce cube sera défini par son centre et son rayon. Les cubes limites à taille fixe sont très faciles à recalculer

[3]réalisé par Kelvin Chung à l'Université de Hong Kong

lorsque les objets se déplacent, et donc conviennent bien pour les environnements dynamiques.

Lors de la phase de pré-traitement, nous calculons le centre et le rayon du cube limite à taille fixe associé à un objet, et nous faisons cela pour tous les objets de l'environnement. Ensuite, à chaque tranche de temps lorsqu'un objet se déplace, nous recalculons le cube associé comme suit :

1. le centre est transformé en utilisant une simple multiplication vecteur-matrice
2. le minimum et maximum suivant les coordonnées x, y et z est calculé en soustrayant et additionnant le rayon des coordonnées du centre (3 additions et 3 soustractions).

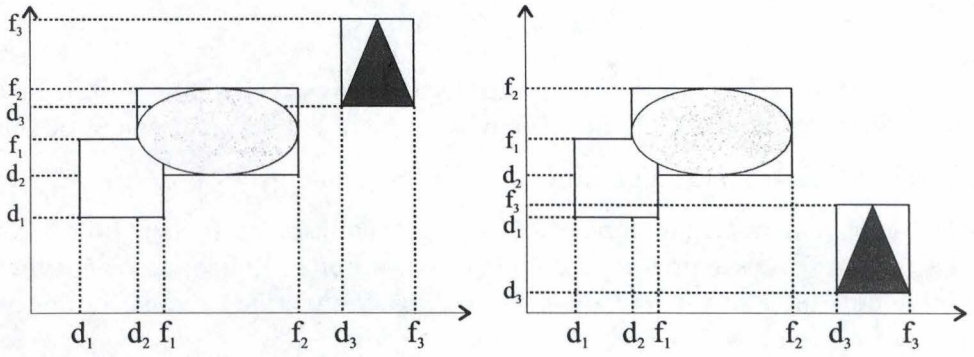
- Boîtes limites rectangulaires dynamiques :
  Il faut calculer la dimension de la boîte limite rectangulaire afin qu'elle soit la boîte limite à axes alignés englobant le plus précisément possible l'objet qui est dans une orientation particulière. Cette boîte est définie par les coordonnées x, y, z minimum et maximum de l'objet. Lorsque l'objet se déplace, nous devons recalculer les minima et maxima de la boîte en tenant compte de l'orientation de l'objet. Pour des objets très allongés, les boîtes limites rectangulaires englobent les objets plus précisément que les cubes. Ce qui implique un nombre moins élevé de volumes limites se chevauchant.

  Dans un environnement virtuel, le gain de temps obtenu en réduisant le nombre de détection de collision entre paire d'objet l'emporte sur le coût de calcul des boîtes limites rectangulaires dynamiques.

  Lors de la phase de pré-traitement, nous calculons pour chaque objet les coordonnées minimum et maximum suivant chacun des axes. Ensuite, à chaque tranche de temps lorsqu'un objet se déplace, nous recalculons la boîte en mettant à jour ces coordonnées minimums et maximums.

Maintenant que nous avons nos boîtes limites englobant les objets, nous allons commencer par projeter chacune des boîtes limites tridimensionnelles sur les axes X, Y et Z. Puisque les boîtes limites sont alignées aux axes, leurs projections donnent comme résultats des intervalles. Nous nous intéressons aux chevauchements parmi ces intervalles, parce qu'une paire de boîtes limites se chevauchent si et seulement si leurs intervalles se chevauchement dans les trois dimensions.

Nous allons construire trois listes, une pour chaque dimension. Chaque liste contient la valeur des points extrémaux des intervalles dans chacune des dimensions correspondantes. En triant ces listes, nous pouvons déterminer quels intervalles se chevauchent. En général, un tel tri à une complexité algorithmique de l'ordre de $O(n\ log\ n)$ où n est le nombre d'objets. Il est possible de réduire ce temps en mémorisant les listes triées de la tranche de temps précédente et en ne changeant que les points extrémaux des intervalles qui ont changé. Dans les environnements où les objets ne font que de petits mouvements entre chacune des images de l'animation, les listes restent presque trié. Donc, nous pouvons utiliser un tri par insertion qui lui a une complexité algorithmique de l'ordre de $O(n)$.



(a) t=1 (b) t=2

FIG. 6.3 – Des boîtes limites se chevauchant dans deux étapes de temps successives (a) $t = 1$ et (b) $t = 2$ (2D)

En plus du tri, nous devons garder la trace des changements du statut de chevauchement des pairs d'intervalles, c'est-à-dire, de se chevauchant dans la tranche de temps précédente vers ne se chevauchant plus dans la tranche de temps courante, et vice-versa. Ce statut de chevauchement consiste en un booléen pour chacune des dimensions. Lorsque ces booléens sont vrais pour chacune des trois dimensions, les boîtes limites associées se chevauchent. Ces booléens sont modifiés uniquement lorsque le tri par insertion effectue une permutation. Il détermine s'il faut ou pas changer la valeur d'un booléen suivant que les valeurs permutées correspondent toutes deux à des minima, toutes deux à des maxima ou une correspond à un maximum et l'autre à un minimum.

Lorsque qu'un booléen est modifié, le statut de chevauchement indique l'une des trois situations suivantes :

1. les intervalles associés à cette paire de boîtes limites se chevauchent. Dans ce cas, nous ajoutons la paire d'objets correspondant aux boîtes limites dans une liste résultat.
2. la paire de boîtes limites se chevauchait dans la tranche de temps précédente. Nous devons donc retirer cette paire d'objets de la liste résultat.
3. la paire de boîtes limites ne se chevauchait pas dans la tranche de temps précédente et ne se chevauche pas dans la courante. Nous ne devons rien faire dans ce cas si.

Lorsque le tri est terminé pour la tranche de temps actuel, la liste résultat contient toutes les paires d'objets dont les boîtes limites associées se chevauchent actuellement. Nous pouvons alors communiquer cette liste à une routine de détection de collision entre paire d'objets.

6.1.3 Approche 2 : plans séparateurs

Nous allons, comme dans l'approche précédente, construire des boîtes limites alignées aux axes englobant chacun des n objets de l'environnement le plus précisément possible. Un fois cette phase de pré-traitement effectuée, nous appliquons sur les $\begin{pmatrix} n \\ 2 \end{pmatrix} = \frac{n(n-1)}{2}$ paires possibles de boîtes limites l'algorithme de détection de collision entre deux objets convexes (section 4.4 page 43). Les paires d'objets ayant leurs boîtes limites associées fortement éloignées l'une de l'autre sont très rapidement éliminées de la liste des paires des d'objets dont les boîtes limites associées se chevauchent actuellement.

Nous pouvons communiquer cette liste à une routine de détection de collision entre paire d'objets.

Pour chacune des tranches de temps suivantes, les boîtes limites associées aux objets sont mises à jour et nous réappliquons le test des plans séparateurs sur les $\frac{n(n-1)}{2}$ paires possibles de boîtes limites.

6.2 Utilisation d'un algorithme pour objets convexes sur des objets concaves

Beaucoup d'algorithmes de détection de collision nécessitent des objets convexes en entrées. Or, il est clair que la plupart des objets intéressants du monde réel sont concaves et un environnement virtuel, pour être utile, devrait accepter des objets concaves.

Il serait fort intéressant d'utiliser un des algorithmes précédents, pour les polyèdres convexes, pour détecter les collisions entre des objets non-convexes. Pour résoudre ce problème, deux approches sont possibles, une avant l'exécution et l'autre pendant l'exécution :

- un objet concave peut être modelé comme une collection de morceaux convexes ou
- il peut être découpé en c parties convexes, et ensuite il faudra tester chacune des c^2 paires de parties possibles pour voir si elles sont en collision avec un autre objet.

Le nombre c de parties convexes peut être assez élevé. Actuellement, le problème du partionnage d'un objet graphique concave en un nombre optimal de parties convexes est NP-difficile. Du fait de la grande complexité des algorithmes de détection de collision pour objets concaves, cette approche est de loin la plus efficace.

Ce genre d'approche a cependant ces inconvénients. Tout d'abord, le partitionnement en parties convexes implique la nécessité d'une représentation alternative en cours d'exécution de l'application. De plus, comme ce partitionnement est réalisé dans la phase de pré-calcul, les applications qui ont besoin de modifier la géométrie des objets pendant la détection de collision ne peuvent utiliser cette approche. Finalement, les algorithmes de partitionnage convexe sont non triviaux à implémenter.

La première approche, utilisant une représentation hiérarchique de l'objet concave créée lors de la modélisation, semble la plus raisonnable. En effet, les inconvénients de la méthode

précédente ne s'y trouvent plus. Les noeuds internes de cette représentation hiérarchique peuvent être des sous-parties convexes ou non, mais toutes les feuilles sont des morceaux convexes de l'objet. Par exemple, un objet concave représentant une main (figure 6.4) est constitué d'articulations se trouvant dans les feuilles, de doigts formant les noeuds internes et la main entière joue le rôle de racine de la hiérarchie.

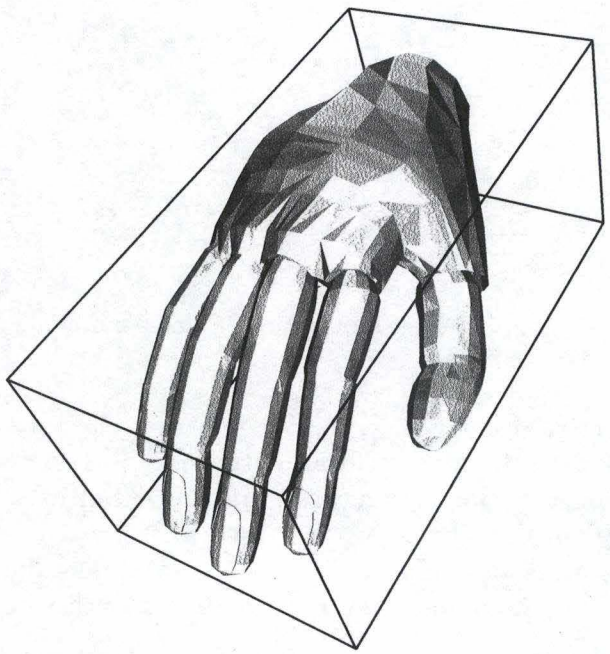


FIG. 6.4 – Exemple d'un objet concave : la représentation d'une main

6.3 Le futur : la détection de collision dynamique

L'un des gros inconvénient des méthodes de détection de collision actuelles, que nous avons vu précédemment, est qu'elles ne sont pas capables de détecter si une collision a eu lieu entre deux étapes de temps successives. En bref, des collisions peuvent leur échapper. Précédemment, nous avions supposé que les tranches de temps étaient suffisamment courtes et nous avions recours à la propriété de cohérence temporelle. Mais que faire lorsque les objets se déplacent vite ou ont des mouvements imprévus.

Prenons deux objets A et B qui se déplacent d'une position au temps $t = 0$ vers une position future au temps $t = 1$ (figure 6.5). Nous voudrions plus que simplement savoir si les deux objets sont en collision ou pas au temps $t = 1$. Nous souhaiterions découvrir si les deux objets sont rentrés en collision durant leur déplacement simultané, même si en $t = 1$ il n'y a aucune collision.

Nous distinguons trois types de détection de collision :

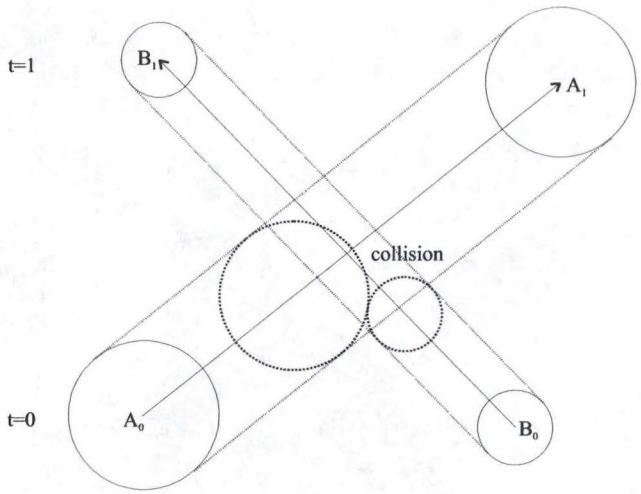


FIG. 6.5 – Exemple de deux sphères en mouvement à deux moments différents t=0 et t=1

- statique : la détection de collision entre des objets est effectuée à un moment donné.
- semi-statique : la phase de test de collision entre des objets est réalisée à des intervalles de temps très court, ceci afin de rater un minimum de collisions. Le tout est de choisir cet intervalle de temps intelligemment.
- dynamique : elle utilise les volumes balayés par les objets durant leur mouvement pour déterminer s'il y a eu une collision entre deux tranches de temps successives.

La détection de collision dynamique considère le mouvement continu des objets. Pour un objet en mouvement et un autre stationnaire qui vont rentrer en collision, cela signifie que non pas l'objet en mouvement lui-même mais son volume balayé est en collision avec l'objet stationnaire.

Le volume balayé par un objet entre deux étapes de temps est assez simple à calculer à partir du moment où nous supposons que le mouvement entre ces deux instants a été rectiligne. Il suffit de calculer l'enveloppe convexe des deux objets dont l'un est celui de l'instant de départ et l'autre celui de l'instant d'arrivé.

Malgré le coût accru de cette nouvelle approche par rapport à l'approche statique, la détection dynamique accepte de plus larges tranches de temps entre chaque test et, bien entendu, sans risque de louper une collision. Les recherches actuelles dans le domaine de la détection de collision se focalisent surtout sur l'aspect dynamique de la détection.

Pour plus d'information sur le sujet de la détection de collision dynamique, nous pouvons nous référencer aux articles [ES99] et [LSW99].

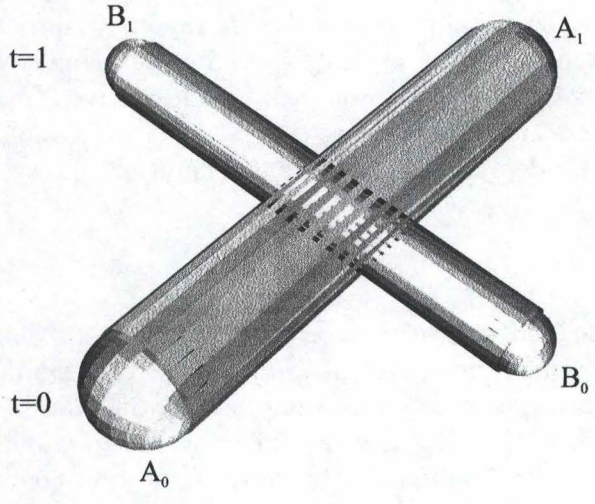


FIG. 6.6 – Les mouvements des deux sphères donnent naissance aux deux nouveaux volumes qui correspondent à l'espace balayé par chacune d'elles

Conclusion et perspectives

La popularité et le succès de l'informatique graphique et des animations placent à un haut niveau le futur de la technologie. Les utilisateurs ne sont plus facilement impressionnables, et ils demandent des effets plus réalistes qu'ils peuvent contrôler plus naturellement et directement. Pour beaucoup d'applications manipulant des objets en mouvement, l'absence de toute forme de détection de collision et de réponse rendra les utilisateurs mal à l'aise. Pour d'autres applications qui présentent des informations de manière visuelle, l'incapacité de manipuler cette information de manière interactive et en temps réel limite ce que peuvent accomplir les utilisateurs. Dans cette section, nous allons résumer les apports de ce mémoire et discuter de quelques-unes des améliorations et extensions possibles.

Conclusion

Nous avons commencé notre investigation en classifiant les diverses questions qu'il est nécessaire de se poser pour avoir une meilleure idée du problème de la détection de collision. Ainsi, il est apparu que la représentation par frontière des objets graphiques était la plus adapté à notre problème. Nous avons vu que les objets graphiques pouvaient avoir des propriétés géométriques différentes (arbitraires, fermés ou convexes) et des comportements différents (stationnaires ou en mouvement, rigides ou déformables). De plus, ce qui est demandé aux algorithmes de détection de collision varie en fonction des besoins des applications graphiques et donc nous pouvions nous attendre à une quantité d'algorithmes de détection de collision adaptés aux besoins des applications et aux objets qu'elles manipulent.

Dans le chapitre 3, nous avons examiné quelques-unes des structures de données les plus couramment utilisées pour représenter les objets polygonaux. Cela nous a permis de mieux comprendre comment est structuré un objet polygonal à l'intérieur de l'ordinateur et ce qu'il est possible de retirer comme informations de ces structures de données.

Notre investigation s'est poursuivie au chapitre 4 où nous avons entamé le problème de la détection de collision entre paire d'objets d'un point de vue algorithmique. Suivant les propriétés géométriques des objets, nous avons constaté qu'il était possible de donner différentes définitions au terme collision. C'est ainsi que nous avons été amenés à analyser les algorithmes de détection de collision les plus appropriés à chacune des définitions. Il s'est avéré que ces algorithmes étaient peu efficaces pour des objets fort volumineux (en

nombre de polygones) et donc il nous fallait chercher une méthode plus appropriée aux objets de grande taille, et si possible arbitraires. C'est ce qui nous a conduits aux méthodes hiérarchiques basées sur les volumes limites. Avant d'entrer dans l'analyse des différents types de hiérarchie, suivant les volumes limites utilisés, nous avons discuté des critères de conception permettant de choisir la hiérarchie la plus appropriée pour une application donnée (en terme de performance).

Nous avons parcouru la plupart des volumes limites existant, en regardant pour chacun comment il est construit, comment l'arbre de volumes limites l'utilisant est construit et comment s'effectue la détection de collision entre deux arbres de ce type.

Nous en avons évalué quatre en tout, en partant des plus simples en allant vers les plus perfectionnés. Les voici avec, s'ils existent, les packages du domaine public les utilisant et les personnes qui les étudient :

Type de volume limite	Nom du package	Personne(s)
boîte limite à axes alignés	SOLID	Gino Van Den Bergen
sphère	-	Philip Hubbard
boîte limite orientée	RAPID	Ming Lin, Stefan Gottschalk, Dinesh Monacha
polytope à orientations discrètes	QuickCD	James T. Klosowski

Nous avons présenté au chapitre 5 une approche matérielle pour la détection de collision entre une paire d'objets. Nous l'avons illustrée avec une application de chirurgie virtuelle. Cette approche met en jeu un objet simple tel qu'une boîte et un autre objet qui peut être de grande taille et totalement arbitraire. Les performances, qui ont été obtenues avec cette approche, sont beaucoup plus élevées qu'avec l'approche algorithmique.

Notre investigation s'est terminée au chapitre 6 par les extensions que nous pouvons apporter à la détection de collision entre paire d'objets. Nous avons commencé par étudier deux méthodes permettant de réduire le problème de détection de collision entre plusieurs objets en un problème entre paire d'objets. Une des méthodes est basée sur l'utilisation d'axes séparateurs tandis que l'autre est basée sur l'utilisation de plans séparateurs. Ensuite, nous nous sommes posé la question de savoir comment utiliser les algorithmes pour des objets graphiques convexes sur des objets graphiques concaves. Cela nous a amenés à envisager deux approches, une avant exécution et une autre pendant exécution. Nous avons constaté que la deuxième approche était peu intéressante puisque le problème de décomposition d'un objet concave en parties convexes est toujours actuellement un problème NP-difficile. Cette approche n'est donc pas envisageable lorsque nous souhaitons obtenir des réponses en temps réel. Finalement, nous nous sommes penchés sur la détection de collision dynamique dont le but est de pallier les insuffisances de la détection de collision statique ou traditionnelle.

Perspectives

Le problème de la détection de collision en temps réel est loin d'être résolu parfaitement et reste toujours un réel goulot d'étranglement pour les applications qui en ont besoin. Il

existe encore beaucoup de voies de recherche que l'on commence seulement à étudier.

Tout d'abord, l'approche matérielle que nous avons vue au chapitre 5 commence seulement à faire l'objet de recherches plus poussées. L'exemple de la chirurgie virtuelle n'est qu'un cas isolé, certains chercheurs envisagent d'utiliser cette méthode pour les tests de chevauchement dans les méthodes hiérarchiques afin de les accélérer grandement.

Ensuite, les récents travaux montrent un grand intérêt pour le problème de la détection de collision dynamique. Les chercheurs se sont rendu compte que l'utilisation permanente de la propriété de cohérence temporelle n'était pas une solution. Dans certain cas, les objets n'obéissent pas toujours de la manière souhaitée. Les recherches actuelles se concentrent sur les méthodes hiérarchiques afin de rendre les algorithmes les utilisant dynamiques.

Finalement, les objets concaves qui représentent une grande majorité des objets graphiques rencontrés ne sont pas laissés de côté. La méthode révolutionnaire qui permettra de les utiliser facilement et rapidement n'est pas encore inventée, mais les recherches se poursuivent.

Annexe A

Terminologie française vs. terminologie anglaise

2-diversifié	2-manifold
arbre octal	octree
arête	edge
boîte limite	bounding box
B-spline rationnelle non-uniforme	NURBS non-uniform rational B-spline
CFAO (Conception et fabrication assistée par ordinateur)	CAD/CAM (computer-assisted design/manufacture)
composant (sommet, arête, polygone)	feature (vertex, edge, polygon)
CSG (géométrie solide constructive)	CSG (constructive solid geometry)
DAO (dessin assisté par ordinateur)	CAD (computer-aided design)

enveloppe convexe	convex hull
gant de données	dataglove
de genre 0	with genus 0
géométrie algorithmique	computational geometry
heure d'arrivée prévue	ETA (estimated time of arrival)
IAO (ingénierie assistée par ordinateur)	CAE (computer-aided engineering)
infographie, informatique graphique	computer graphics
lancer de rayon	ray tracing
lieu(x) géométrique	locus (loci)
polyèdre(s)	polyhedron (polyhedra)
représentation par frontière	B-rep (boundary representation)
retour d'effort	force feedback
sommet(s)	vertex (vertices)
sommet(s) [d'un cône, d'un tétraèdre]	apex (apices)
treillis de polygone	polygon mesh

Annexe B

Les superquadriques superellipsoïdes et supertoroïdes

B.1 Superellipses et Superellipsoïdes

B.1.1 Superellipses

En deux dimensions, une superellipse centrée à l'origine est définie sous sa forme paramétrique par

$$\begin{cases} x = & r_x \cos^n \phi \\ y = & r_y \sin^n \phi \end{cases}$$

$$\text{où } 0 \leq \phi \leq 2\pi \text{ et } 0 < n < \infty$$

La courbe intersecte l'axe des x aux points r_x et $-r_x$, l'axe des y en r_y et $-r_y$. La superellipse est donc entièrement contenue dans un rectangle de largeur $2r_x$ et de hauteur $2r_y$. C'est la forme la plus facile à utiliser pour dessiner une superellipse, en effet, l'angle ϕ est varié de 0 à 2π par petite incrémentation et une ligne est tracée entre chaque valeur successive de ϕ.

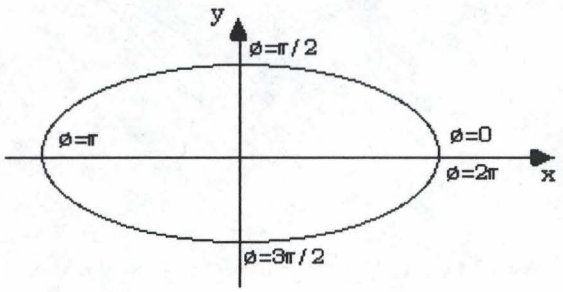


FIG. B.1 – Une superellipse centrée à l'origine

Nous pouvons également écrire

$$f(x,y) = (\frac{x}{r_x})^{\frac{2}{n}} + (\frac{y}{r_y})^{\frac{2}{n}}$$

Ce qui implique que
- si $f(x,y) = 1$ alors (x,y) se trouve sur la frontière de la superellipse ;
- si $f(x,y) < 1$ alors (x,y) se trouve à l'intérieur de la superellipse ;
- si $f(x,y) > 1$ alors (x,y) se trouve à l'extérieur de la superellipse.

En faisant varier la valeur du paramètre n, nous pouvons contrôler aisément la forme de la superellipse comme on peut le voir sur la figure B.2.

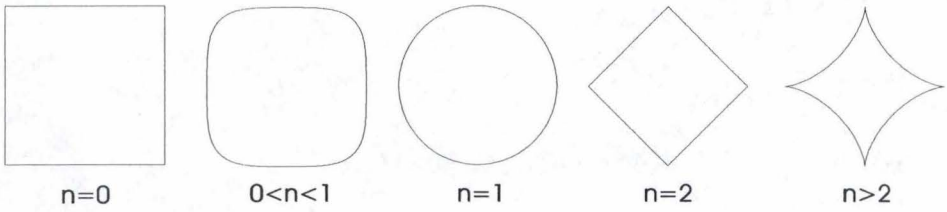


FIG. B.2 – Une superellipse suivant divers valeurs du paramètre n (0, 0.5, 1, 2, 3)

Nous pouvons remarquer qu'une superellipse passe toujours par un même ensemble de points, c'est-à-dire en $\phi = 0$, $\frac{\pi}{2}$, π, $3\frac{\pi}{2}$, et ce quelle que soit la valeur du paramètre n.

B.1.2 Superellipsoïdes

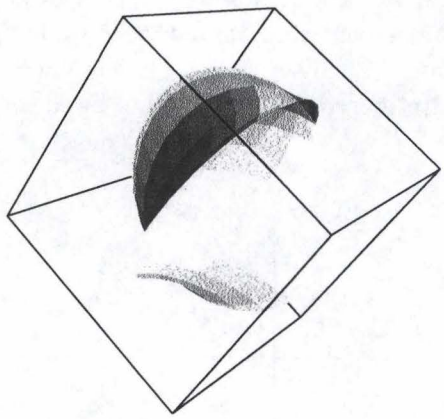


FIG. B.3 – Une superellispsoïde dont les paramètres n_1 et n_2 valent respectivement 3 et 1

En trois dimensions, une superellipse centrée à l'origine est définie sous sa forme para-

métrique par

$$\begin{cases} x = & r_x \cos^{n_1} \alpha \cos^{n_2} \beta \\ y = & r_y \cos^{n_1} \alpha \sin^{n_2} \beta \\ z = & r_z \sin^{n_1} \alpha \end{cases}$$

$$\text{où } -\frac{\pi}{2} \leq \alpha \leq \frac{\pi}{2}, \ -\pi \leq \beta \leq \pi \text{ et } 0 < n_1, n_2 < \infty$$

Comme pour le cas en deux dimensions, r_x, r_y et r_z sont les facteurs d'échelle de chaque axe. Le superellipsoïde obtenu se trouve au centre d'une boîte de dimension $2r_x$, $2r_y$, $2r_z$.

Nous pouvons également écrire

$$f(x,y,z) = [(\frac{x}{r_x})^{\frac{2}{n_2}} + (\frac{y}{r_y})^{\frac{2}{n_2}}]^{\frac{n_2}{n_1}} + (\frac{z}{r_z})^{\frac{2}{n_1}}$$

Nous pouvons tirer les mêmes relations que pour le cas bidimentionnel (superellipse), en effet,

- si $f(x,y,z) = 1$ alors (x,y,z) se trouve sur la frontière du superellipsoïde ;
- si $f(x,y,z) < 1$ alors (x,y,z) se trouve à l'intérieur du superellipsoïde ;
- si $f(x,y,z) > 1$ alors (x,y,z) se trouve à l'extérieur du superellipsoïde.

B.1.3 Code source C

Voici le code source C d'un algorithme qui permet de générer les facettes d'une superellipse dans un fichier au format DXF avec comme paramètre $n_1 = 1.0$ et $n_2 = 0.2$.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct {
  double x,y,z;
} XYZ;

/* pi/180 (pour convertir les radians en degrés) */
#define PI180 0.01745329252

void WriteDXFFacet(FILE *,XYZ *,int);
double power(double,double);

void main()
{
  int u,v,du=15,dv=15;
  double rx=1.0,ry=1.0, rz=1.0;
  double n1=1.0,n2=0.2;
  double theta,phi;
  XYZ p[4];
  FILE *fptr;

  if ((fptr = fopen("ellipse.dxf","w")) == NULL) {
```

```
    fprintf(stderr,"Unable to open DXF file\n");
    return;
  }
  fprintf(fptr,"999\nModel of a superellipsoid\n");
  fprintf(fptr,"0\nSECTION\n");
  fprintf(fptr,"2\nENTITIES\n");

  for (u=-90;u<90;u+=du) {
    printf("theta = %d\n",u);
    for (v=-180;v<180;v+=dv) {
      theta = (u) * PI180;
      phi   = (v) * PI180;
      p[0].x = rx*power(cos(theta),n1) * power(cos(phi),n2);
      p[0].y = ry*power(cos(theta),n1) * power(sin(phi),n2);
      p[0].z = rz * power(sin(theta),n1);
      theta = (u+du) * PI180;
      phi   = (v) * PI180;
      p[1].x = rx*power(cos(theta),n1) * power(cos(phi),n2);
      p[1].y = ry*power(cos(theta),n1) * power(sin(phi),n2);
      p[1].z = rz * power(sin(theta),n1);
      theta = (u+du) * PI180;
      phi   = (v+dv) * PI180;
      p[2].x = rx*power(cos(theta),n1) * power(cos(phi),n2);
      p[2].y = ry*power(cos(theta),n1) * power(sin(phi),n2);
      p[2].z = rz * power(sin(theta),n1);
      theta = (u) * PI180;
      phi   = (v+dv) * PI180;
      p[3].x = rx*power(cos(theta),n1) * power(cos(phi),n2);
      p[3].y = ry*power(cos(theta),n1) * power(sin(phi),n2);
      p[3].z = rz * power(sin(theta),n1);
      WriteDXFFacet(fptr,p,4);
    }
  }

  fprintf(fptr,"0\nENDSEC\n");
  fprintf(fptr,"0\nEOF\n");
  fclose(fptr);
}

void WriteDXFFacet(fptr,p,n)
FILE *fptr;
XYZ *p;
int n;
{
  int i;

  fprintf(fptr,"0\nPOLYLINE\n");
  fprintf(fptr,"8\nsuperellipsoid\n");
  fprintf(fptr,"66\n1\n");
  fprintf(fptr,"70\n9\n");
  for (i=0;i<n;i++) {
```

```
    fprintf(fptr,"0\nVERTEX\n");
    fprintf(fptr,"8\nsuperellipsoid\n");
    fprintf(fptr,"70\n32\n");
    fprintf(fptr,"10\n%g\n",p[i].x);
    fprintf(fptr,"20\n%g\n",p[i].y);
    fprintf(fptr,"30\n%g\n",p[i].z);
  }
  fprintf(fptr,"0\nSEQEND\n");
}

double power(f,p)
double f,p;
{
  int sign;
  double absf;

  sign = (f < 0 ? -1 : 1);
  absf = (f < 0 ? -f : f);

  if (absf < 0.00001)
    return(0.0);
  else
    return(sign * pow(absf,p));
}
```

B.2 Supertoroïdes

Regardons à présent un type d'objet tout aussi intéressant que les superellipsoïdes et faisant aussi partie des superquadriques. Le supertoroïde est une famille de primitives géométriques basée sur le tore. Soient r_0 le rayon du disque interne et r_1 le rayon du disque externe, comme illustré sur la figure B.6

L'équation paramétrique d'un supertoroïde est la même que celle du tore excepté que les termes sinus et cosinus sont élevé à la puissance n_1 ou n_2.

$$\begin{cases} x = & \cos^{n_1}\theta(r_0 + r_1\cos^{n_2}\phi) \\ y = & \sin^{n_1}\theta(r_0 + r_1\cos^{n_2}\phi) \\ z = & r_1\sin^{n_2}\phi \end{cases}$$

où θ et ϕ varient de 0 à 2π.

La valeur de n_1 détermine la forme de l'anneau du tore tandis que n_2 détermine la forme de la section transversale de l'anneau.

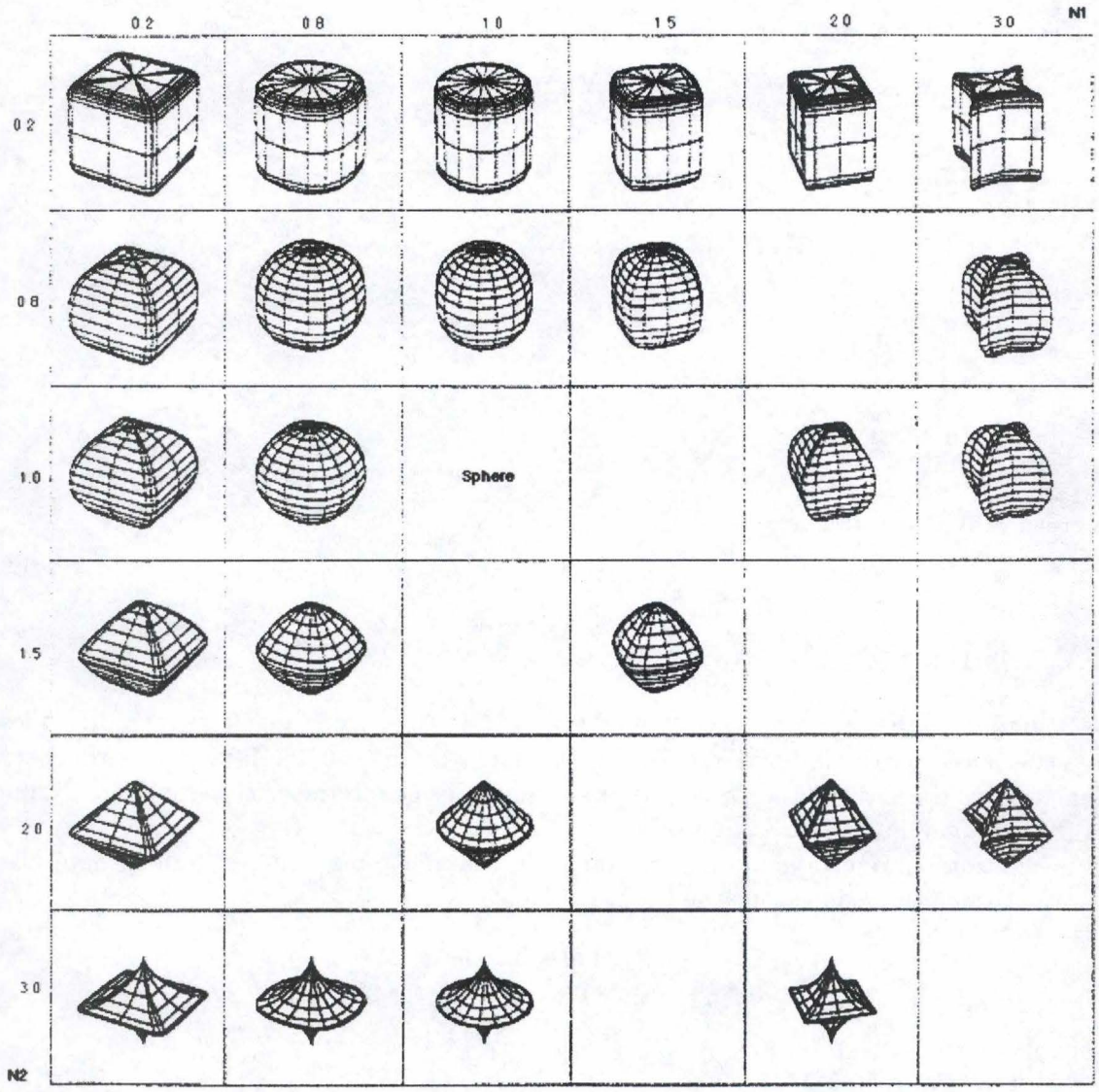


FIG. B.4 – Les différentes formes de superellipsoïdes suivant les valeurs des paramètres n_1 et n_2

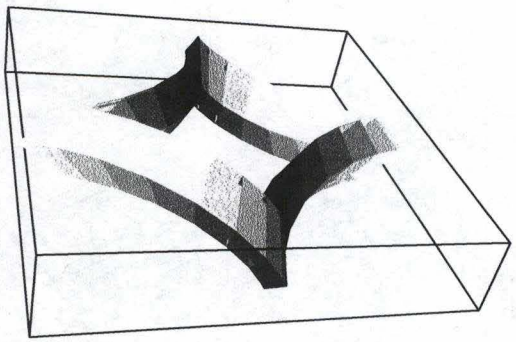


FIG. B.5 – Une superellispsoïde dont les paramètres n_1 et n_2 valent respectivement 3 et 2

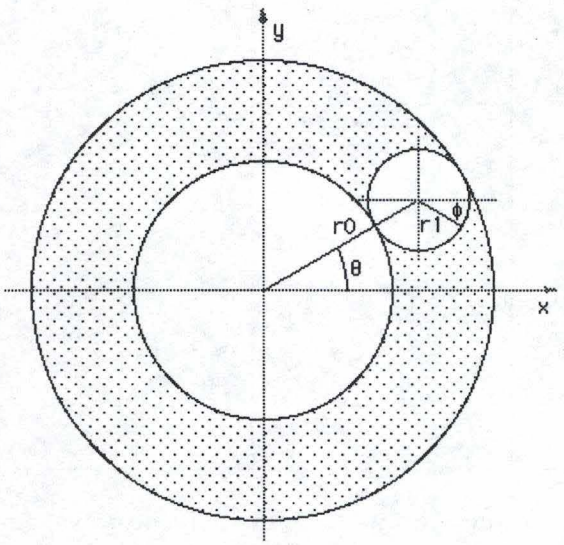


FIG. B.6 – Coupe transversale d'un supertoroïde où θ et ϕ varient de 0 à 2π

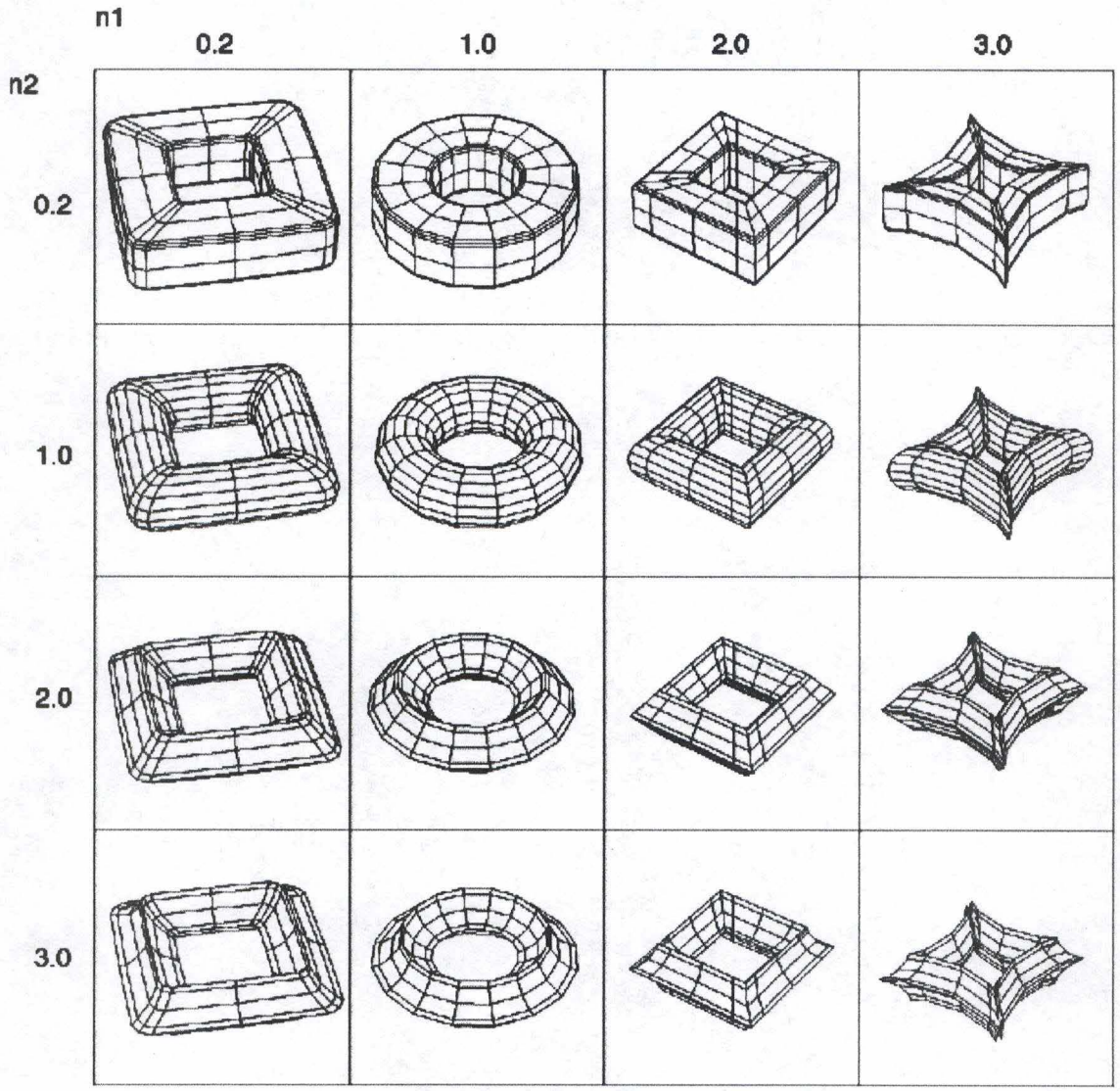


FIG. B.7 – Les différentes formes de supertoroïdes suivant les valeurs des paramètres n_1 et n_2

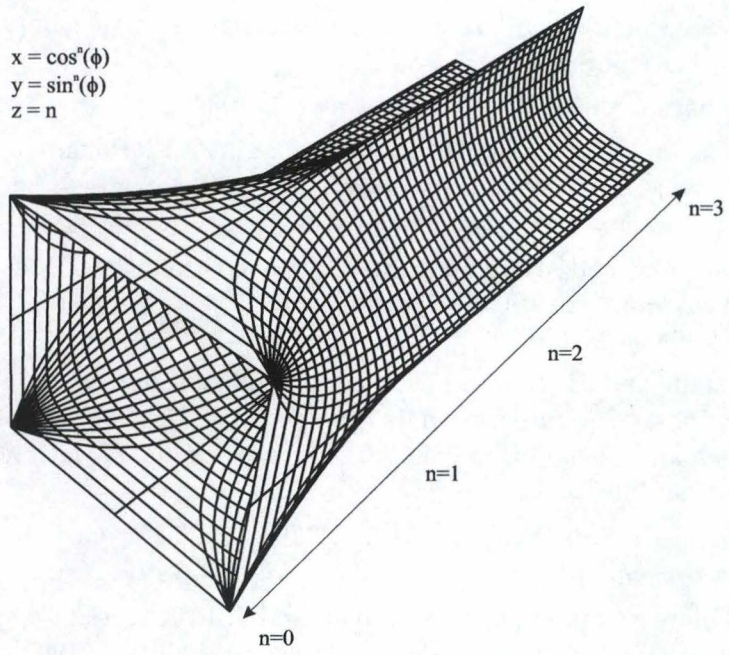


FIG. B.8 – La section transversale de l'anneau du tore lorsque n varie de 0 à 3

Bibliographie

[Bau72] B. Baumgart. Winged–edge polyhedron representation. Technical Report CS–320, Dept. of Computer Science, Stanford University, Stanford, CA, 1972.

[Ber98] Gino Van Den Bergen. Efficient collision detection of complex deformables models. Novembre 1998.

[Bou89] P. D. Bourke. Vision-3d. Technical report, 1989.

[CLMP95] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-collide : An interactive and exact collision detection system for large-scale environments. *1995 Symposium on Interactive 3D Graphics*, pages 189–196, Avril 1995.

[CW96] Kelvin Chung and Wenping Wang. Quick collision detection of polytopes in virtual environments. *ACM Symposium on Virtual Reality Software and Technology 1996*, Juillet 1996.

[ES99] J. Eckstein and E. Schömer. Dynamic collision detection in virtual reality applications. *7th International Conference in Central Europe on Computer Graphics and Visualization and Interactive Digital Media, WSCG'99*, pages 71–78, 1999.

[FvDF+97] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, and R.L. Phillips. *Introduction to computer graphics.* Addison-Westley, 1997.

[FvDFH93] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practices.* Addison-Westley, 2nd edition, 1993.

[GG99] J. Grad and S. Grad. Génération de vues 3d de paysages. Master's thesis, Facultés Universitaires Notre-Dame de la Paix, Département de Mathématique, Mai 1999.

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. Obb-tree : A hierarchical structure for rapid interference detection. *Proceedings of SIGGRAPH 96*, pages 171–180, Août 1996.

[Got] Stefan Gottschalk. Collision detection techniques for 3d models.

[Got96] Stefan Gottschalk. Separating axis theorem. Technical report, Dept. of Computer Science, UNC Chapel Hill, 1996.

[HBZ90] Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. Geometric collisions for time-dependent parametric surfaces. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4) :39–48, Août 1990.

[He99] Taosong He. Fast collision detection using quospo trees. *ACM Symposium on Interactive 3D Graphics*, Avril 1999.

[Hub95] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3) :218–230, Septembre 1995.

[KHM+88] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), Mars 1988.

[LC92] Ming C. Lin and John F. Canny. Efficient collision detection for animation. *Third Eurographics Workshop on Animation and Simulation*, Septembre 1992.

[LCF99] J.-C. Lombardo, M.-P. Cani, and F.Neyret. Real-time collision detection for virtual surgery. In *Computer Animation*, Geneva Switzerland, May 26-28 1999.

[Lin93] M. C. Lin. *Efficient Collision Detection for Animation and Robotics.* PhD thesis, University of California at Berkeley, Departement of Electrical Engineering and Computer Science, Decembre 1993.

[LM95] Ming C. Lin and Dinesh Manocha. Fast interference detection between geometric models. *The Visual Computer*, 11(10) :542–551, 1995.

[LSW99] C. Lennerz, E. Schömer, and T. Warken. A framework for collision detection and response. *11th European Simulation Symposium, ESS'99*, pages 309–314, 1999.

[Mir98] Brian Mirtich. V-clip : Fast and robust polyhedral collision detection. *ACM Trans. on Graphics*, 17(3) :177–208, Juillet 1998.

[MW88] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. *Computer Graphics (Proceedings of SIGGRAPH 88)*, 22(4) :289–298, Août 1988.

[O'R98] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 2nd edition, 1998.

[PML97] Madhav K. Ponamgi, Dinesh Manocha, and Ming C. Lin. Incremental algorithms for collision detection between polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(1) :51–64, Janvier - Mars 1997.

[PTVF97] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, The Art of Scientific Computing.* Cambridge University Press, 2nd edition, 1997.

[Sch99] P. Schorn. Xyz geobench manual v5.0.1. Technical report, Institut für Theoretische Informatik, Zürich, Août 1999.

[Toi95] Ph.L. Toint. Algèbre. *Département de Mathématique, Facultés Universitaires Notre-Dame de la Paix*, Septembre 1995.

[Wat98] A. Watt. *3D Computer Graphics.* Addison-Westley, 2nd edition, 1998.

[WNDS99] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL programming guide : the official guide to learning OpenGL version 1.2.* Addison-Westley, 3rd edition, Septembre 1999.

[Zac97] Gabriel Zachmann. Real-time and exact collision detection for interactive virtual protyping. *Proc. of the 1997 ASME Design Engineering Technical Conferences*, Septembre 1997.

[ZPOM93] M. J. Zyda, D. R. Pratt, W. D. Osborne, and J. G. Monahan. Npsnet : Real-time collision detection and response. *The Journal of Visualization and Computer Animation*, 4(1) :13–24, Janvier - Mars 1993.